Capítulo 2

ORDENACIÓN

2.1 INTRODUCCIÓN

Dado un conjunto de n elementos $a_1, a_2,..., a_n$ y una relación de orden total (≤) sobre ellos, el problema de la ordenación consiste en encontrar una permutación de esos elementos ordenada de forma creciente.

Aunque tanto el tipo y tamaño de los elementos como el dispositivo en donde se encuentran almacenados pueden influir en el método que utilicemos para ordenarlos, en este tema vamos a solucionar el caso en que los elementos son números enteros y se encuentran almacenados en un vector.

Si bien existen distintos criterios para clasificar a los algoritmos de ordenación, una posibilidad es atendiendo a su eficiencia. De esta forma, en función de la complejidad que presentan en el caso medio, podemos establecer la siguiente clasificación:

- $\Theta(n^2)$: Burbuja, Inserción, Selección.
- $\Theta(n\log n)$: Mezcla, Montículos, Quicksort.
- Otros: Incrementos $\Theta(n^{1.25})$, Cubetas $\Theta(n)$, Residuos $\Theta(n)$.

En el presente capítulo desarrollaremos todos ellos con detenimiento, prestando especial atención a su complejidad, no sólo en el caso medio sino también en los casos mejor y peor, pues para algunos existen diferencias significativas. Hemos dedicado también una sección a problemas, que recogen muchas de las cuestiones y variaciones que se plantean durante el estudio de los distintos métodos.

Como hemos mencionado anteriormente, nos centraremos en la ordenación de enteros, muchos de los problemas de ordenación que nos encontramos en la práctica son de ordenación de registros mucho más complicados. Sin embargo este problema puede ser fácilmente reducido al de ordenación de números enteros utilizando las claves de los registros, o bien índices. Por otro lado, puede que los datos a ordenar excedan la capacidad de memoria del ordenador, y por tanto deban residir en dispositivos externos. Aunque este problema, denominado *ordenación externa*, presenta ciertas dificultades específicas (véase [AHO87]), los métodos utilizados para resolverlo se basan fundamentalmente en los algoritmos que aquí presentamos.

Antes de pasar a desarrollar los principales algoritmos, hemos considerado necesario precisar algunos detalles de implementación.

- Consideraremos que el tamaño máximo de la entrada y el vector a ordenar vienen dados por las siguientes definiciones:

```
CONST n =...; (* numero maximo de elementos *)
TYPE vector = ARRAY [1..n] OF INTEGER;
```

- Los procedimientos de ordenación que presentamos en este capítulo están diseñados para ordenar cualquier subvector de un vector dado $a[1..n]$. Por eso generalmente poseerán tres parámetros: el nombre del vector que contiene a los elementos (a) y las posiciones de comienzo y fin del subvector, como por ejemplo *Seleccion(a,prim,ult)*. Para ordenar todo el vector, basta con invocar al procedimiento con los valores $prim = 1$, $ult = n$.
- Haremos uso de dos funciones que permiten determinar la posición de los elementos máximo y mínimo de un subvector dado:

```
PROCEDURE PosMaximo(VAR a:vector;i,j:CARDINAL):CARDINAL;
(* devuelve la posicion del maximo elemento de a[i..j] *)
   VAR pmax,k:CARDINAL;
BEGIN
   pmax:=i;
   FOR k:=i+1 TO j DO
      IF a[k]>a[pmax] THEN
         pmax:=k
      END
   END;
   RETURN pmax;
END PosMaximo;

PROCEDURE PosMinimo(VAR a:vector;i,j:CARDINAL):CARDINAL;
(* devuelve la posicion del minimo elemento de a[i..j] *)
   VAR pmin,k:CARDINAL;
BEGIN
   pmin:=i;
   FOR k:=i+1 TO j DO
      IF a[k]<a[pmin] THEN
         pmin:=k
      END
   END;
   RETURN pmin;
END PosMinimo;
```

- Y utilizaremos un procedimiento para intercambiar dos elementos de un vector:

```
PROCEDURE Intercambia(VAR a:vector;i,j:CARDINAL);
(* intercambia a[i] con a[j] *)
   VAR temp:INTEGER;
BEGIN
   temp:=a[i];
   a[i]:=a[j];
   a[j]:=temp
END Intercambia;
```

Veamos los tiempos de ejecución de cada una de ellas:

a) El tiempo de ejecución de la función *PosMaximo* va a depender, además del tamaño del subvector de entrada, de su ordenación inicial, y por tanto distinguiremos tres casos:

– En el caso mejor, la condición del *IF* es siempre falsa. Por tanto:

$$T(n) = T(j-i+1) = 1 + \left(\left(\sum_{k=i+1}^{j}(3+3)\right) + 3\right) + 1 = 5 + 6(j-i).$$

– En el caso peor, la condición del *IF* es siempre verdadera. Por consiguiente:

$$T(n) = T(j-i+1) = 1 + \left(\left(\sum_{k=i+1}^{j}(3+3+1)\right) + 3\right) + 1 = 5 + 7(j-i).$$

– En el caso medio, vamos a suponer que la condición del *IF* será verdadera en el 50% de los casos. Por tanto:

$$T(n) = T(j-i+1) = 1 + \left(\left(\sum_{k=i+1}^{j}(3+3+\frac{1}{2})\right) + 3\right) + 1 = 5 + \frac{13}{2}(j-i).$$

Estos casos corresponden respectivamente a cuando el elemento máximo se encuentra en la primera posición, en la última y el vector está ordenado de forma creciente, o cuando consideramos equiprobables cada una de las *n* posiciones en donde puede encontrarse el máximo.

Como podemos apreciar, en cualquiera de los tres casos su complejidad es lineal con respecto al tamaño de la entrada.

b) El tiempo de ejecución de la función *PosMinimo* es exactamente igual al de la función *PosMaximo*.

c) La función *Intercambia* realiza 7 operaciones elementales (3 asignaciones y 4 accesos al vector), independientemente de los datos de entrada.

Nótese que en las funciones *PosMaximo* y *PosMinimo* hemos utilizado el paso del vector *a* por referencia en vez de por valor (mediante el uso de *VAR*) para evitar la copia del vector en la pila de ejecución, lo que incrementaría la complejidad del algoritmo resultante, pues esa copia es de orden O(n).

2.2 ORDENACIÓN POR INSERCIÓN

El método de Inserción realiza n–1 iteraciones sobre el vector, dejando en la i-ésima etapa ($2 \leq i \leq n$) ordenado el subvector $a[1..i]$. La forma de hacerlo es colocando en cada iteración el elemento $a[i]$ en su sitio correcto, aprovechando el hecho de que el subvector $a[1..i-1]$ ya ha sido previamente ordenado. Este método puede ser implementado de forma iterativa como sigue:

```
PROCEDURE Insercion(VAR a:vector;prim,ult:CARDINAL);
   VAR i,j:CARDINAL; x:INTEGER;
BEGIN
   FOR i:=prim+1 TO ult DO
      x:=a[i]; j:=i-1;
      WHILE (j>=prim) AND (x<a[j]) DO
         a[j+1]:=a[j]; DEC(j)
      END;
      a[j+1]:=x
   END
END Insercion;
```

Para estudiar su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Insercion(a,*1*,n)*.

– En el caso mejor el bucle interno no se realiza nunca, y por tanto:

$$T(n) = \left(\sum_{i=2}^{n} (3+4+4+3) \right) + 3 = 14n - 11 .$$

– En el caso peor hay que llevar cada elemento hasta su posición final, con lo que el bucle interno se realiza siempre de i–1 veces. Así, en este caso:

$$T(n) = \left(\sum_{i=2}^{n} \left(3+4+\left(\sum_{j=1}^{i-1} (4+5) \right) + 1 + 3 \right) \right) + 3 = \frac{9}{2}n^2 + \frac{13}{2}n - 10 .$$

– En el caso medio, supondremos equiprobable la posición de cada elemento dentro del vector. Por tanto para cada valor de i, la probabilidad de que el elemento se sitúe en alguna posición k de las i primeras será de $1/i$. El número de veces que se repetirá el bucle *WHILE* en este caso es (i–k), con lo cual el número medio de operaciones que se realizan en el bucle es:

$$\left(\frac{1}{i}\sum_{k=1}^{i} 9(i-k)\right) + 4 = \frac{9}{2}i - \frac{1}{2}.$$

Por tanto, el tiempo de ejecución en el caso medio es:

$$T(n) = \left(\sum_{i=2}^{n}\left(3+4+\left(\frac{9}{2}i-\frac{1}{2}\right)+3\right)\right)+3 = \frac{9}{4}n^2 + \frac{47}{4}n - 11.$$

Por el modo en que funciona el algoritmo, tales casos van a corresponder a cuando el vector se encuentra ordenado de forma creciente, decreciente o aleatoria.

Como podemos ver, en este método los órdenes de complejidad de los casos peor, mejor y medio difieren bastante. Así en el mejor caso el orden de complejidad resulta ser lineal, mientras que en los casos peor y medio su complejidad es cuadrática.

Este método se muestra muy adecuado para aquellas situaciones en donde necesitamos ordenar un vector del que ya conocemos que está casi ordenado, como suele suceder en aquellas aplicaciones de inserción de elementos en bancos de datos previamente ordenados cuya ordenación total se realiza periódicamente.

2.3 ORDENACIÓN POR SELECCIÓN

En cada paso (i=1...n–1) este método busca el mínimo elemento del subvector $a[i..n]$ y lo intercambia con el elemento en la posición i:

```
PROCEDURE Seleccion(VAR a:vector;prim,ult:CARDINAL);
   VAR i:CARDINAL;
BEGIN
   FOR i:=prim TO ult-1 DO
      Intercambia(a,i,PosMinimo(a,i,ult))
   END
END Seleccion;
```

En cuanto a su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Seleccion(a,*1*,n)*, que van a coincidir con los mismos casos (mejor, peor y medio) que los de la función *PosMinimo*.

– En el caso mejor:

$$T(n) = \left(\sum_{i=1}^{n-1}(3+1+(5+6(n-i))+1+7)\right)+3 = 3n^2 + 14n - 14.$$

– En el caso peor:

$$T(n) = \left(\sum_{i=1}^{n-1} \big(3 + 1 + \big(5 + 7(n-i)\big) + 1 + 7\big) \right) + 3 = \frac{7}{2}n^2 + \frac{27}{2}n - 14.$$

– En el caso medio:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + 1 + \left(5 + \frac{13}{2}(n-i) \right) + 1 + 7 \right) \right) + 3 = \frac{13}{4}n^2 + \frac{55}{4}n - 14.$$

En consecuencia, el algoritmo es de complejidad cuadrática.

Este método, por el número de operaciones de comparación e intercambio que realiza, es el más adecuado para ordenar pocos registros de gran tamaño. Si el tipo base del vector a ordenar no es entero, sino un tipo más complejo (guías telefónicas, índices de libros, historiales hospitalarios, etc.) deberemos darle mayor importancia al intercambio de valores que a la comparación entre ellos en la valoración del algoritmo por el coste que suponen. En este sentido, analizando el número de intercambios que realiza el método de Selección vemos que es de orden O(n), frente al orden O(n^2) de intercambios que presentan los métodos de Inserción o Burbuja.

2.4 ORDENACIÓN BURBUJA

Este método de ordenación consiste en recorrer los elementos siempre en la misma dirección, intercambiando elementos adyacentes si fuera necesario:

```
PROCEDURE Burbuja (VAR a:vector;prim,ult:CARDINAL);
   VAR i,j:CARDINAL;
BEGIN
   FOR i:=prim TO ult-1 DO
      FOR j:=ult TO i+1 BY -1 DO
         IF (a[j-1]>a[j]) THEN
            Intercambia(a,j-1,j)
         END
      END
   END
END Burbuja;
```

El nombre de este algoritmo trata de reflejar cómo el elemento mínimo "sube", a modo de burbuja, hasta el principio del subvector.

Respecto a su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Burbuja(a,*1*,n)*.

– En el caso mejor:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + \sum_{j=i+1}^{n} (3+4) + 3 \right) \right) + 3 = \frac{7}{2}n^2 + \frac{5}{2}n - 3.$$

– En el caso peor:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + \sum_{j=i+1}^{n} (3+4+2+7) + 3 \right) \right) + 3 = 8n^2 - 2n - 1.$$

– En el caso medio:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + \sum_{j=i+1}^{n} \left(3+4+\frac{2+7}{2} \right) + 3 \right) \right) + 3 = \frac{23}{4}n^2 + \frac{1}{4}n - 1.$$

En consecuencia, el algoritmo es de complejidad cuadrática.

Este algoritmo funciona de forma parecida al de Selección, pero haciendo más trabajo para llevar cada elemento a su posición. De hecho es el peor de los tres vistos hasta ahora, no sólo en cuanto al tiempo de ejecución, sino también respecto al número de comparaciones y de intercambios que realiza.

Una posible mejora que puede admitir este algoritmo es el control de la existencia de una pasada sin intercambios; en ese momento el vector estará ordenado.

2.5 ORDENACIÓN POR MECLA (MERGESORT)

Este método utiliza la técnica de Divide y Vencerás para realizar la ordenación del vector *a*. Su estrategia consiste en dividir el vector en dos subvectores, ordenarlos mediante llamadas recursivas, y finalmente combinar los dos subvectores ya ordenados. Esta idea da lugar a la siguiente implementación:

```
PROCEDURE Mezcla(VAR a,b:vector;prim,ult:CARDINAL);
(* utiliza el vector b como auxiliar para realizar la mezcla *)
   VAR mitad:CARDINAL;
BEGIN
   IF prim<ult THEN
      mitad:=(prim+ult)DIV 2;
      Mezcla(a,b,prim,mitad);
      Mezcla(a,b,mitad+1,ult);
      Combinar(a,b,prim,mitad,mitad+1,ult)
   END
END Mezcla;
```

Una posible implementación de la función que lleva a cabo el proceso de mezcla vuelca primero los elementos a ordenar en el vector auxiliar para después, utilizando dos índices, uno para cada subvector, rellenar el vector ordenadamente. Nótese que el algoritmo utiliza el hecho de que los dos subvectores están ya ordenados y que son además consecutivos.

```
PROCEDURE Combinar(VAR a,b:vector;p1,u1,p2,u2:CARDINAL);
(* mezcla ordenadamente los subvectores a[p1..u1] y a[p2..u2]
   suponiendo que estos estan ya ordenados y que son consecutivos
   (p2=u1+1), utilizando el vector auxiliar b *)
   VAR i1,i2,k:CARDINAL;
BEGIN
   IF (p1>u1) OR (p2>u2) THEN RETURN END;
   FOR k:=p1 TO u2 DO b[k]:=a[k] END; (* volcamos a en b *)
   i1:=p1;i2:=p2;  (* cada indice se encarga de un subvector *)
   FOR k:=p1 TO u2 DO
      IF b[i1]<=b[i2] THEN
         a[k]:=b[i1];
         IF i1<u1 THEN INC(i1) ELSE b[i1]:=MAX(INTEGER) END
      ELSE
         a[k]:=b[i2];
         IF i2<u2 THEN INC(i2) ELSE b[i2]:=MAX(INTEGER) END
      END
   END
END Combinar;
```

En cuanto al estudio de su complejidad, siguiendo el mismo método que hemos utilizado en los problemas del primer capítulo, se llega a que el tiempo de ejecución de *Mezcla(a,b,*1*,n)* puede expresarse mediante una ecuación en recurrencia:

$$T(n) = 2T(n/2) + 16n + 17$$

con la condición inicial $T(1) = 1$. Ésta es una ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x–2)^2(x–1) = 0$, lo que permite expresar $T(n)$ como:

$$T(n) = c_1n + c_2n\log n + c_3.$$

El cálculo de las constantes puede hacerse en base a la condición inicial, lo que nos lleva a la expresión final:

$$T(n) = 16n\log n + 18n – 17 \in \Theta(n\log n).$$

Obsérvese que este método ordena n elementos en tiempo $\Theta(n\log n)$ en cualquiera de los casos (peor, mejor o medio). Sin embargo tiene una complejidad espacial, en cuanto a memoria, mayor que los demás (del orden de n).

Otras versiones de este algoritmo no utilizan el vector auxiliar *b*, sino que trabajan sobre el propio vector a ordenar, combinando sobre él los subvectores

obtenidos de las etapas anteriores. Si bien es cierto que esto consigue ahorrar espacio (un vector auxiliar), también complica el código del algoritmo resultante.

El método de ordenación por Mezcla se adapta muy bien a distintas circunstancias, por lo que es comúnmente utilizado no sólo para la ordenación de vectores. Por ejemplo, el método puede ser también implementado de forma que el acceso a los datos se realice de forma secuencial, por lo que hay diversas estructuras (como las listas enlazadas) para las que es especialmente apropiado. También se utiliza para realizar ordenación externa, en donde el vector a ordenar reside en dispositivos externos de acceso secuencial (i.e. ficheros).

2.6 ORDENACIÓN MEDIANTE MONTÍCULOS (HEAPSORT)

La filosofía de este método de ordenación consiste en aprovechar la estructura particular de los montículos (heaps), que son árboles binarios completos (todos sus niveles están llenos salvo a lo sumo el último, que se rellena de izquierda a derecha) y cuyos nodos verifican la propiedad del montículo: todo nodo es mayor o igual que cualquiera de sus hijos. En consecuencia, en la raíz se encuentra siempre el elemento mayor.

Estas estructuras admiten una representación muy sencilla, compacta y eficiente mediante vectores (por ser árboles completos). Así, en un vector que represente una implementación de un montículo se cumple que el "padre" del *i*-ésimo elemento del vector se encuentra en la posición $i \div 2$ (menos la raíz, claro) y sus "hijos", si es que los tiene, estarán en las posiciones $2i$ y $2i+1$ respectivamente.

La idea es construir, con los elementos a ordenar, un montículo sobre el propio vector. Una vez construido el montículo, su elemento mayor se encuentra en la primera posición del vector (*a*[*prim*]). Se intercambia entonces con el último (*a*[*ult*]) y se repite el proceso para el subvector *a*[*prim..ult*–1]. Así sucesivamente hasta recorrer el vector completo. Esto nos lleva a un algoritmo de orden de complejidad O(*n*log*n*) cuya implementación puede ser la siguiente:

```
PROCEDURE Monticulos(VAR a:vector;prim,ult:CARDINAL);
   VAR i:CARDINAL;
BEGIN
   HacerMonticulo(a,prim,ult);
   FOR i:=ult TO prim+1 BY -1 DO
      Intercambia(a,prim,i);
      Empujar(a,prim,i-1,prim)
   END
END Monticulos;
```

Los procedimientos *HacerMontículo* y *Empujar* son, respectivamente, el que construye un montículo a partir del subvector *a*[*prim..ult*] dado, y el que "empuja" un elemento hasta su posición definitiva en el montículo, reconstruyendo la estructura de montículo en el subvector *a*[*prim..ult*–1]:

```
PROCEDURE HacerMonticulo(VAR a:vector;prim,ult:CARDINAL);
(* construye un monticulo a partir de a[prim..ult] *)
   VAR i:CARDINAL;
BEGIN
   FOR i:=(ult-prim+1)DIV 2 TO 1 BY -1 DO
      Empujar(a,prim,ult,prim+i-1)
   END
END HacerMonticulo;


PROCEDURE Empujar(VAR a:vector;prim,ult,i:CARDINAL);
(* empuja el elemento en posicion i hasta su posicion final *)
   VAR j,k:CARDINAL;
BEGIN
   k:=i-prim+1;
   REPEAT
      j:=k;
      IF (2*j<=ult-prim+1) AND (a[2*j+prim-1]>a[k+prim-1]) THEN
         k:=2*j
      END;
      IF (2*j<ult-prim+1) AND (a[2*j+prim]>a[k+prim-1]) THEN
         k:=2*j+1
      END;
      Intercambia(a,j+prim-1,k+prim-1);
   UNTIL j=k
END Empujar;
```

Para estudiar la complejidad del algoritmo hemos de considerar dos partes. La primera es la que construye inicialmente el montículo a partir de los elementos a ordenar y la segunda va recorriendo en cada iteración un subvector más pequeño, colocando el elemento raíz en su posición correcta dentro del montículo. En ambos casos nos basamos en la función que "empuja" elementos en el montículo.

Observando el comportamiento del algoritmo, la diferencia básica entre el caso peor y el mejor está en la profundidad que hay que recorrer cada vez que necesitamos "empujar" un elemento. Si el elemento es menor que todos los demás, necesitaremos recorrer todo el árbol (profundidad: $\log n$); si el elemento es mayor o igual que el resto, no será necesario.

El procedimiento *HacerMonticulo* es de complejidad $O(n)$ en el peor caso, puesto que si k es la altura del montículo ($k = \log n$), el algoritmo transforma primero cada uno de los dos subárboles que cuelgan de la raíz en montículos de altura a lo más $k-1$ (el subárbol derecho puede tener altura $k-2$), y después empuja la raíz hacia abajo, por un camino que a lo más es de longitud k. Esto lleva a lo más un tiempo $t(k)$ de orden de complejidad $O(k)$ con lo cual

$$T(k) \leq 2T(k-1) + t(k),$$

ecuación en recurrencia cuya solución verifica que $T(k) \in O(2^k)$. Como $k = \log n$, la complejidad de *HacerMonticulo* es lineal en el peor caso. Este caso ocurre cuando

hay que recorrer siempre la máxima profundidad al empujar a cada elemento, lo que sucede si el vector está originalmente ordenado de forma creciente.

Respecto al mejor caso de *HacerMonticulo*, éste se presenta cuando la profundidad a la que hay que empujar cada elemento es cero. Esto se da, por ejemplo, si todos los elementos del vector son iguales. En esta situación la complejidad del algoritmo es O(1).

Estudiemos ahora los casos mejor y peor del resto del algoritmo *Montículos*. En esta parte hay un bucle que se ejecuta siempre n–1 veces, y la complejidad de la función que intercambia dos elementos es O(1). Todo va a depender del procedimiento *Empujar*, es decir, de la profundidad a la que haya que empujar la raíz del montículo en cada iteración, sabiendo que cada montículo tiene n–i elementos, y por tanto una altura de log(n–i), siendo i el número de la iteración.

En el peor caso, la profundidad a la que hay que empujar las raíces respectivas es la máxima, y por tanto la complejidad de esta segunda parte del algoritmo es O($n$log$n$). ¿Cuándo ocurre esto? Cuando el elemento es menor que todos los demás. Pero esto sucede siempre que los elementos a ordenar sean distintos, por la forma en la que se van escogiendo las nuevas raíces.

En el caso mejor, aunque el bucle se sigue repitiendo n–1 veces, las raíces no descienden, por ser mayores o iguales que el resto de los elementos del montículo. Así, la complejidad de esta parte del algoritmo es de orden O(n). Pero este caso sólo se dará si los elementos del vector son iguales, por la forma en la que originariamente se construyó el montículo y por cómo se escoge la nueva raíz en cada iteración (el último de los elementos, que en un montículo ha de ser de los menores).

2.7 ORDENACIÓN RÁPIDA DE HOARE (QUICKSORT)

Este método es probablemente el algoritmo de ordenación más utilizado, pues es muy fácil de implementar, trabaja bien en casi todas las situaciones y consume en general menos recursos (memoria y tiempo) que otros métodos.

Su diseño está basado en la técnica de Divide y Vencerás, que estudiaremos en el siguiente capítulo, y consta de dos partes:

a) En primer lugar el vector a ordenar *a*[*prim..ult*] es dividido en dos subvectores no vacíos *a*[*prim..l*–1] y *a*[*l*+1..*ult*], tal que todos los elementos del primero son menores que los del segundo. El elemento de índice *l* se denomina *pivote* y se calcula como parte del procedimiento de partición.

b) A continuación, los dos subvectores son ordenados mediante llamadas recursivas a Quicksort. Como los subvectores se ordenan sobre ellos mismos, no es necesario realizar ninguna operación de combinación.

Esto da lugar al siguiente procedimiento, que constituye la versión clásica del algoritmo de ordenación rápida de Hoare:

```
PROCEDURE Quicksort(VAR a:vector;prim,ult:CARDINAL);
   VAR l:CARDINAL;
BEGIN
   IF prim<ult THEN
      l:=Pivote(a,a[prim],prim,ult);
      Quicksort(a,prim,l-1);
      Quicksort(a,l+1,ult)
   END
END Quicksort;
```

La función *Pivote* parte del elemento pivote y permuta los elementos del vector de forma que al finalizar la función, todos los elementos menores o iguales que el pivote estén a su izquierda, y los elementos mayores que él a su derecha. Devuelve la posición en la que ha quedado situado el pivote *p*:

```
PROCEDURE Pivote(VAR a:vector;p:INTEGER;prim,ult:CARDINAL)
                                                    :CARDINAL;
  (* permuta los elementos de a[prim..ult] y devuelve una
     posicion l tal que prim<=l<=ult, a[i]<=p si prim<=i<l,
     a[l]=p, y a[i]>p si l<i<=ult, donde p es el valor inicial
     de a[prim] *)
   VAR i,l:CARDINAL;
BEGIN
   i:=prim; l:=ult+1;
   REPEAT INC(i) UNTIL (a[i]>p) OR (i>=ult);
   REPEAT DEC(l) UNTIL (a[l]<=p);
   WHILE i<l DO
      Intercambia(a,i,l);
      REPEAT INC(i) UNTIL (a[i]>p);
      REPEAT DEC(l) UNTIL (a[l]<=p)
   END;
   Intercambia(a,prim,l);
   RETURN l
END Pivote;
```

Este método es de orden de complejidad $\Theta(n^2)$ en el peor caso y $\Theta(n\log n)$ en los casos mejor y medio. Para ver su tiempo de ejecución, utilizando los mecanismos expuestos en el primer capítulo, obtenemos la siguiente ecuación en recurrencia:

$$T(n) = 8 + T(a) + T(b) + T_{Pivote}(n)$$

donde *a* y *b* son los tamaños en los que la función *Pivote* divide al vector (por tanto podemos tomar que $a + b = n$), y $T_{Pivote}(n)$ es la función que define el tiempo de ejecución de la función *Pivote*.

El procedimiento *Quicksort* “rompe” la filosofía de caso mejor, peor y medio de los algoritmos clásicos de ordenación, pues aquí tales casos no dependen de la ordenación inicial del vector, sino de la elección del pivote.

Así, el mejor caso ocurre cuando $a = b = n/2$ en todas las invocaciones recursivas del procedimiento, pues en este caso obtenemos $T_{Pivote}(n) = 13 + 4n$, y por tanto:

$$T(n) = 21 + 4n + 2T(n/2).$$

Resolviendo esta ecuación en recurrencia y teniendo en cuenta las condiciones iniciales $T(0) = 1$ y $T(1) = 27$ se obtiene la expresión final de $T(n)$, en este caso:

$$T(n) = 15n\log n + 26n + 1.$$

Ahora bien, si $a = 0$ y $b = n–1$ (o viceversa) en todas las invocaciones recursivas del procedimiento, $T_{Pivote}(n) = 11 + 39/8n$, obteniendo:

$$T(n) = 19 + 39/8n + T(n–1).$$

Resolviendo la ecuación para las mismas condiciones iniciales, nos encontramos con una desagradable sorpresa:

$$T(n) = \frac{3}{8}n^2 + \frac{213}{8}n + 1 \in \Theta(n^2).$$

En consecuencia, la elección idónea para el pivote es la mediana del vector en cada etapa, lo que ocurre es que encontrarla requiere un tiempo extra que hace que el algoritmo se vuelva más ineficiente en la mayoría de los casos (ver problema 2.15). Por esa razón como pivote suele escogerse un elemento cualquiera, a menos que se conozca la naturaleza de los elementos a ordenar. En nuestro caso, como a priori suponemos equiprobable cualquier ordenación inicial del vector, hemos escogido el primer elemento del vector, que es el que se le pasa como segundo argumento a la función *Pivote*.

Esta elección lleva a tres casos desfavorables para el algoritmo: cuando los elementos son todos iguales y cuando el vector está inicialmente ordenado en orden creciente o decreciente. En estos casos la complejidad es cuadrática puesto que la partición se realiza de forma totalmente descompensada.

A pesar de ello suele ser el algoritmo más utilizado, y se demuestra que su tiempo promedio es menor, en una cantidad constante, al de todos los algoritmos de ordenación de complejidad $O(n\log n)$. En todo esto es importante hacer notar, como hemos indicado antes, la relevancia que toma una buena elección del pivote, pues de su elección depende considerablemente el tiempo de ejecución del algoritmo.

Sobre el algoritmo expuesto anteriormente pueden realizarse varias mejoras:

1. Respecto a la elección del pivote. En vez de tomar como pivote el primer elemento, puede seguirse alguna estrategia del tipo:

    - Tomar al azar tres elementos seguidos del vector y escoger como pivote el elemento medio de los tres.
    - Tomar k elementos al azar, clasificarlos por cualquier método, y elegir el elemento medio como pivote.

2. Con respecto al tamaño de los subvectores a ordenar. Cuando el tamaño de éstos sea pequeño (menor que una cota dada), es posible utilizar otro algoritmo de ordenación en vez de invocar recursivamente a Quicksort. Esta idea utiliza el hecho de que algunos métodos, como Selección o Inserción, se comportan muy bien cuando el número de datos a ordenar son pocos, por disponer de constantes multiplicativas pequeñas. Aun siendo de orden de complejidad cuadrática, son más eficientes que los de complejidad $n\log n$ para valores pequeños de n.

En los problemas propuestos y resueltos se desarollan más a fondo estas ideas.

2.8 ORDENACIÓN POR INCREMENTOS (SHELLSORT)

La ordenación por inserción puede resultar lenta pues sólo intercambia elementos adyacentes. Así, si por ejemplo el elemento menor está al final del vector, hacen falta n pasos para colocarlo donde corresponde. El método de Incrementos es una extensión muy simple y eficiente del método de Inserción en el que cada elemento se coloca *casi* en su posición definitiva en la primera pasada.

El algoritmo consiste básicamente en dividir el vector a en h subvectores:

$$a[k],\ a[k+h],\ a[k+2h],\ a[k+3h],\ ...$$

y ordenar por inserción cada uno de esos subvectores (k=1,2,...,h–1).

Un vector de esta forma, es decir, compuesto por h subvectores ordenados intercalados, se denomina h-ordenado. Haciendo h-ordenaciones de a para valores grandes de h permitimos que los elementos puedan moverse grandes distancias dentro del vector, facilitando así las h-ordenaciones para valores más pequeños de h. A h se le denomina incremento.

Con esto, el método de ordenación por Incrementos consiste en hacer h-ordenaciones de a para valores de h decreciendo hasta llegar a uno.

El número de comparaciones que se realizan en este algoritmo va a depender de la secuencia de incrementos h, y será mayor que en el método clásico de Inserción (que se ejecuta finalmente para $h = 1$), pero la potencia de este método consiste en conseguir un número de intercambios mucho menor que con la Inserción clásica.

El procedimiento presentado a continuación utiliza la secuencia de incrementos h = ..., 1093, 364, 121, 40, 13, 1. Otras secuencias pueden ser utilizadas, pero la elección ha de hacerse con cuidado. Por ejemplo la secuencia ..., 64, 32, 16, 8, 4, 2, 1 es muy ineficiente pues los elementos en posiciones pares e impares no son comparados hasta el último momento. En el ejercicio 2.7 se discute más a fondo esta circunstancia.

```
PROCEDURE Incrementos(VAR a:vector;prim,ult:CARDINAL);
   VAR i,j,h,N:CARDINAL; v:INTEGER;
BEGIN
   N:=(ult-prim+1);  (* numero de elementos *)
   h:=1;
```

```
    REPEAT h:=3*h+1 UNTIL h>N;  (* construimos la secuencia *)
    REPEAT
      h:=h DIV 3;
      FOR i:=h+1 TO N DO
        v:=a[i]; j:=i;
        WHILE (j>h) AND (a[j-h+prim-1]>v) DO
          a[j+prim-1]:=a[j-h+prim-1];
          DEC(j,h)
        END;
        a[j+prim-1]:=v;
      END
    UNTIL h=1
  END Incrementos;
```

En cuanto al estudio de su complejidad, este método es diferente al resto de los procedimientos vistos en este capítulo. Su complejidad es difícil de calcular y depende mucho de la secuencia de incrementos que utilice. Por ejemplo, para la secuencia dada existen dos conjeturas en cuanto a su orden de complejidad: $n\log^2 n$ y $n^{1.25}$. En general este método es el escogido para muchas aplicaciones reales por ser muy simple teniendo un tiempo de ejecución aceptable incluso para grandes valores de n.

2.9 OTROS ALGORITMOS DE ORDENACIÓN

Los algoritmos vistos hasta ahora se basan en la ordenación de vectores de números enteros cualesquiera, sin ningún tipo de restricción. En este apartado veremos cómo pueden encontrarse algoritmos de orden O(n) cuando dispongamos de información adicional sobre los valores a ordenar.

2.9.1 Ordenación por Cubetas (Binsort)

Suponemos que los datos a ordenar son números naturales, todos distintos y comprendidos en el intervalo [1,n]. Es decir, nuestro problema es ordenar un vector con los n primeros números naturales. Bajo esas circunstancias es posible implementar un algoritmo de complejidad temporal O(n). Es el método de ordenación por Cubetas, en donde en cada iteración se sitúa un elemento en su posición definitiva:

```
  PROCEDURE Cubetas(VAR a:vector);
    VAR i:CARDINAL;
  BEGIN
    FOR i:=1 TO n DO
      WHILE a[i]<>i DO
```

```
            Intercambia(a,i,a[i])
          END
        END
      END Cubetas;
```

2.9.2 Ordenación por Residuos (Radix)

Este método puede utilizarse cuando los valores a ordenar están compuestos por secuencias de letras o dígitos que admiten un orden lexicográfico. Éste es el caso de palabras, números (cuyos dígitos admiten este orden) o bien fechas.

El método consiste en definir *k* colas (numeradas de 0 a *k*–1) siendo *k* los posibles valores que puede tomar cada uno de los dígitos que componen la secuencia. Una vez tengamos las colas habría que repetir, para *i* a partir de 0 y hasta llegar al número máximo de dígitos o letras de nuestras cadenas:

1. Distribuir los elementos en las colas en función del dígito *i*.
2. Extraer ordenada y consecutivamente los elementos de las colas, introduciéndolos de nuevo en el vector.

Los elementos quedan ordenados sin haber realizado ninguna comparación. Veamos un ejemplo de este método. Supongamos el vector:

[0, 1, 81, 64, 23, 27, 4, 25, 36, 16, 9, 49].

En este caso se trata de números naturales en base 10, que no son sino secuencias de dígitos. Como cada uno de los dígitos puede tomar 10 valores (del 0 al 9), necesitaremos 10 colas. En la primera pasada introducimos los elementos en las colas de acuerdo a su dígito menos significativo:

Cola	0	1	2	3	4	5	6	7	8	9
	0	81,1		23	4,64	25	16,36	27		49,9

y ahora extraemos ordenada y sucesivamente los valores, obteniendo el vector:

[0, 81, 1, 23, 4, 64, 25, 16, 36, 27, 49, 9].

Volvemos a realizar otra pasada, esta vez fijándonos en el segundo dígito menos significativo:

Cola	0	1	2	3	4	5	6	7	8	9
	9,4,1,0	16	27,25,23	36	49		64		81	

Volviendo a extraer ordenada y sucesivamente los valores obtenemos el vector [0, 1, 4, 9, 16, 23, 25, 27, 36, 49, 64, 81]. Como el máximo de dígitos de los números a ordenar era de dos, con dos pasadas hemos tenido suficiente.

La implementación de este método queda resuelta en el problema 2.9, en donde se discute también su complejidad espacial, inconveniente principal de estos métodos tan eficientes.

2.10 PROBLEMAS PROPUESTOS

2.1. En los algoritmos Burbuja o Selección, el elemento más pequeño de $a[i..n]$ es colocado en la posición i mediante intercambios, para valores sucesivos de i. Otra posibilidad es colocar el elemento máximo de $a[1..j]$ en la posición j, para valores de j entre n y 1. A este algoritmo se le denomina ordenación por Ladrillos (Bricksort). Implementar dicho algoritmo y estudiar su complejidad.

2.2. Una variante curiosa de los algoritmos anteriores resulta al combinar los métodos de la Burbuja y de los Ladrillos. La idea es ir colocando alternativamente el mayor valor de $a[1..j]$ en $a[j]$, y el menor valor de $a[i..n]$ en $a[i]$. Implementar dicho algoritmo, conocido como Sacudidas (Shakersort), y estudiar su complejidad.

2.3. Modificar el algoritmo de ordenación por Selección de forma que se intercambien los elementos únicamente si son distintos. ¿Qué impacto tiene esta modificación sobre la complejidad del algoritmo?

2.4. Modificar los algoritmos Quicksort y Mezcla de forma que sustituyan las llamadas recursivas por llamadas al procedimiento Selección cuando el tamaño del vector a ordenar sea menor que una cota dada M.

2.5. ¿Cuándo se presentan en el método de ordenación mediante Montículos el mejor y el peor caso?

2.6. Realizar implementaciones iterativas para los procedimientos Quicksort y Mezcla. Estudiar sus complejidades (espacio y tiempo) y comparar los resultados con los obtenidos para las versiones recursivas.

2.7. Para el algoritmo de ordenación por Incrementos, dar un ejemplo que muestre que 2^k,...,8,4,2,1 no es una buena secuencia de incrementos.

2.8. En el método de ordenación Quicksort, ¿qué ocurre si todos los elementos son iguales? ¿Cómo puede modificarse el algoritmo para optimizar este caso especial?

2.9. Implementar el método *Residuos* para ordenar números naturales a partir de su representación (*a*) decimal y (*b*) binaria. Estudiar detalladamente las complejidades de los algoritmos resultantes.

2.10. Un algoritmo de ordenación se denomina *estable* si, dados dos elementos con claves iguales, después de ordenarlos tienen el mismo orden que tenían antes de la clasificación. La estabilidad es importante cuando un vector ha sido ya ordenado por una clave y necesita ser ordenado por otra. Averiguar cuales de los métodos siguientes son estables y cuales no: Selección, Inserción, Burbuja, Incrementos, Quicksort, Mezcla, Montículos, Ladrillos

y Sacudidas. Para aquellos que no lo sean, dar un ejemplo que corrobore la afirmación y proponer modificaciones al método para convertirlo en estable.

2.11. Modificar el método de Inserción de manera que use la búsqueda binaria para localizar dónde introducir el siguiente elemento. Estudiar el impacto de esta mejora en la complejidad del algoritmo y decidir si es rentable o no.

2.12. El algoritmo de ordenación por Rastreo de un vector funciona de la siguiente manera: comienza por el principio del vector y se mueve hacia el final del vector, comparando los pares de elementos adyacentes hasta encontrar uno que no esté en orden correcto. Lo intercambia y comienza a moverse hacia el principio, intercambiando pares hasta encontrar un par en el orden correcto. Entonces se limita a cambiar de dirección y comienza otra vez hacia el final del vector, buscando de nuevo un par fuera de orden. Una vez que alcanza el extremo final del vector, su misión ha terminado. Implementar dicho algoritmo y calcular su tiempo de ejecución en los casos mejor, peor y medio.

2.13. En el método de ordenación por Mezcla, en vez de dividir el vector $a[1..n]$ en dos mitades, podríamos dividirlo en tres subvectores de tamaños $n÷3$, $(n+1)÷3$ y $(n+2)÷3$, ordenarlos recursivamente, y luego combinarlos. Implementar este algoritmo, calcular su complejidad y compararlo con Mezcla.

2.14. Supongamos el siguiente procedimiento:

```
PROCEDURE OrdenarTres(VAR a:vector;prim,ult:CARDINAL);
   VAR k:CARDINAL; temp:INTEGER;
BEGIN
   IF a[prim]>a[ult] THEN
      temp:=a[prim]; a[prim]:=a[ult]; a[ult]:=temp
   END;
   IF prim+1>=ult THEN RETURN END;
   k:=(ult-prim+1)DIV 3;
   OrdenarTres(a,prim,ult-k); (* primeros 2/3 *)
   OrdenarTres(a,prim+k,ult); (* ultimos  2/3 *)
   OrdenarTres(a,prim,ult-k); (* otra vez los primeros 2/3 *)
END OrdenarTres;
```

Calcular el tiempo de ejecución de *OrdenarTres(a,*1*,n)* en función de n y su orden de complejidad, comparándolo con los otros métodos de ordenación.

2.15. *El problema del k-ésimo elemento*: Dado un vector de enteros, queremos encontrar el elemento que ocuparía la posición k si el vector estuviera ordenado en orden creciente (esto es, el k-ésimo menor elemento). Una primera idea para resolver este problema consiste en ordenar primero el vector y después escoger el elemento en la posición k, pero la complejidad de este algoritmo es $O(n\log n)$. ¿Puede hacerse de alguna forma más eficiente? Considerar las dos siguientes ideas y comparar sus complejidades:

- Ordenar el vector sólo hasta la posición k, utilizando un método incremental como el de Selección.
- Utilizar un procedimiento basado en la idea de Quicksort, escogiendo como pivote el elemento en la posición k del vector.

2.16. Un vector contiene n elementos. Se desea encontrar los m elementos más pequeños del vector, con $m<n$. Indicar cuál de las siguientes opciones es la mejor, justificando la respuesta:

a) Ordenar el vector entero y escoger los m primeros elementos.

b) Ordenar los m primeros elementos del vector usando repetidamente el procedimiento de Selección.

c) Invocar m veces al procedimiento que encuentra el k-ésimo elemento (problema 2.15), con los subvectores apropiados.

d) Mediante otro método.

2.17. Supongamos un vector como en el problema anterior, pero ahora queremos encontrar los elementos que ocuparían las posiciones $n\div2$, $(n\div2)+1,...,(n\div2)+m-1$ si el vector estuviese ordenado. Indicar cuál de las siguientes opciones es la mejor, justificando la respuesta:

a) Ordenar el vector entero y escoger los m elementos indicados.

b) Ordenar los elementos apropiados del vector usando repetidamente el procedimiento de Selección.

c) Invocar m veces al procedimiento que encuentra el k-ésimo elemento (problema 2.15), con los subvectores apropiados.

d) Mediante otro método.

2.11 SOLUCIÓN A LOS PROBLEMAS PROPUESTOS

Solución al Problema 2.1. (☺)

Haciendo uso de las funciones presentadas en la introducción de este capítulo, el procedimiento de ordenación por Ladrillos puede ser implementado como sigue:

```
PROCEDURE Ladrillos(VAR a:vector;prim,ult:CARDINAL);
   VAR j:CARDINAL;
BEGIN
   FOR j:=ult TO prim+1 BY -1 DO
      Intercambia(a,j,PosMaximo(a,prim,j))
   END
END Ladrillos;
```

En cuanto a su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Ladrillos(a,*1*,n)*, que van a coincidir con los mismos casos (mejor, peor y medio) que los de la función *PosMaximo*.

– En el caso mejor:

$$T(n) = \left(\sum_{i=2}^{n} \big(3+1+\big(5+6(i-1)\big)+1+7\big) \right) + 3 = 3n^2 + 14n - 14.$$

– En el caso peor:

$$T(n) = \left(\sum_{i=2}^{n} \big(3+1+\big(5+7(i-1)\big)+1+7\big) \right) + 3 = \frac{7}{2}n^2 + \frac{27}{2}n - 14.$$

– En el caso medio:

$$T(n) = \left(\sum_{i=2}^{n} \left(3+1+\left(5+\frac{13}{2}(i-1)\right)+1+7 \right) \right) + 3 = \frac{13}{4}n^2 + \frac{55}{4}n - 14.$$

En consecuencia, el algoritmo es de complejidad cuadrática.

Solución al Problema 2.2. (☺)

El procedimiento que realiza la ordenación por Sacudidas puede ser implementado colocando alternativamente en su posición definitiva el máximo y el mínimo de un subvector cada vez más pequeño, como muestra el siguiente algoritmo:

```
PROCEDURE Sacudidas(VAR a:vector;prim,ult:CARDINAL);
BEGIN
   WHILE prim<ult DO
      Intercambia(a,ult,PosMaximo(a,prim,ult));
      DEC(ult);
      Intercambia(a,prim,PosMinimo(a,prim,ult));
      INC(prim)
   END
END Sacudidas;
```

Para estudiar su complejidad, vamos a fijarnos en la llamada al procedimiento *Sacudidas(a,*1*,n)*. Sus casos mejor, peor y medio corresponden a los de las funciones que encuentran el máximo y el mínimo del vector. El valor de $T(n)$ resulta ser:

– En el mejor caso:

$$\left(\sum_{k=1}^{n/2}\big(1+\big(1+\big(6(2k-1)+5\big)+1+7\big)+1+\big(1+\big(6(2k-2)+5\big)+1+7\big)+1\big)\right)+1=$$

$$=3n^2+\frac{25}{2}n+1.$$

– En el peor caso:

$$\left(\sum_{k=1}^{n/2}\big(1+\big(1+\big(7(2k-1)+5\big)+1+7\big)+1+\big(1+\big(7(2k-2)+5\big)+1+7\big)+1\big)\right)+1=$$

$$=\frac{7}{2}n^2+12n+1.$$

– En el caso medio:

$$\left(\sum_{k=1}^{n/2}\left(1+\left(1+\left(\frac{13}{2}(2k-1)+5\right)+1+7\right)+1+\left(1+\left(\frac{13}{2}(2k-2)+5\right)+1+7\right)+1\right)\right)+1$$

$$=\frac{13}{4}n^2+\frac{49}{4}n+1.$$

Por consiguiente, podemos concluir que el algoritmo Sacudidas es de complejidad cuadrática.

Solución al Problema 2.3. (☺)

Observando la implementación realizada del procedimiento *Selección* al principio del capítulo podemos ver que en él se intercambian elementos adyacentes independientemente de que sean iguales o no. Podemos realizar la modificación

pedida del algoritmo preguntando antes de cada intercambio si es necesario, obteniendo:

```
PROCEDURE Seleccion2(VAR a:vector;prim,ult:CARDINAL);
   VAR i,j:CARDINAL;
BEGIN
   FOR i:=prim TO ult-1 DO
      j:=PosMinimo(a,i,ult);
      IF a[i]<>a[j] THEN
         Intercambia(a,i,j)
      END
   END
END Seleccion2;
```

Para el cálculo de su complejidad, vamos a considerar sus casos mejor, peor y medio de la llamada al procedimiento *Seleccion2(a,*1*,n)*:

– En el caso mejor el elemento mínimo se encuentra en la primera posición. En consecuencia, la condición es siempre falsa. En este caso:

$$T(n) = \left(\sum_{i=1}^{n-1} \big(3 + 6(n-i) + 5 + 1 + 1 + 3\big) \right) + 3 = 3n^2 + 10n - 10 .$$

– En el caso peor el elemento máximo se encuentra en la última posición, y la condición es siempre verdadera. Así, en este caso:

$$T(n) = \left(\sum_{i=1}^{n-1} \big(3 + 7(n-i) + 5 + 1 + 1 + 3 + 1 + 7\big) \right) + 3 = \frac{7}{2}n^2 + \frac{35}{2}n - 18 .$$

– En el caso medio, el elemento mínimo puede estar de forma equiprobable en cualquiera de las posiciones del subvector en el que lo buscamos, y supondremos además que la condición se verifica la mitad de las veces. Por tanto:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + \frac{13}{2}(n-i) + 5 + 1 + 1 + 3 + \frac{(1+7)}{2} \right) \right) + 3 = \frac{13}{4}n^2 + \frac{55}{4}n - 14 .$$

Nótese que los casos mejor y peor corresponden por tanto a las situaciones en donde el vector a ordenar está ya ordenado o cuando está ordenado en sentido inverso.

Para comparar los dos algoritmos, lo primero es observar que ambos son de complejidad cuadrática y, más aún, que los límites de los cocientes de las funciones *T*(*n*) en los tres casos valen exactamente 1, lo que implica que en los tres casos los algoritmos convergen asintóticamente de la misma forma.

En cualquier caso, y cara a afinar más el análisis, podemos restar los tiempos de ejecución de ambos algoritmos en los tres casos y estudiar el signo y la magnitud de la diferencia, obteniendo:

a) En el caso mejor $T_{Seleccion}(n) - T_{Seleccion2}(n) = -4(n - 1)$.
b) En el caso peor $T_{Seleccion}(n) - T_{Seleccion2}(n) = -4(n - 1)$.
c) En el caso medio $T_{Seleccion}(n) - T_{Seleccion2}(n) = 0$.

Como puede observarse, el algoritmo modificado (*Seleccion2*) es un poco mejor en dos de los tres casos. Sin embargo, las diferencias encontradas no son demasiado significativas.

Solución al Problema 2.4. (☺)

Supongamos que disponemos de una cota dada *M*:

```
CONST M =...;
```

que indica el número de elementos mínimo a partir del cual Quicksort debe invocar al procedimiento *Seleccion*. Es fácil modificar el algoritmo para tener en cuenta este hecho:

```
PROCEDURE Quicksort2(VAR a:vector;prim,ult:CARDINAL);
   VAR l:CARDINAL;
BEGIN
   IF prim<ult THEN
      IF ult-prim<M THEN
         Seleccion(a,prim,ult)
      ELSE
         l:=Pivote(a,a[prim],prim,ult);
         Quicksort2(a,prim,l-1);
         Quicksort2(a,l+1,ult)
      END
   END
END Quicksort2;
```

El procedimiento *Seleccion* es la que implementa el método de ordenación por Selección, expuesta al comienzo de este capítulo.

Primero analiza el caso base dado (que el vector tenga menos de *M* elementos), terminando su ejecución tras ordenar el vector en ese caso. Si el vector a ordenar tiene más de los elementos indicados, el algoritmo continúa como antes.

Dada ahora la constante *M*, la modificación al procedimiento *Mezcla* no plantea tampoco mayor dificultad:

```
PROCEDURE Mezcla2(VAR a,b:vector;prim,ult:CARDINAL);
```

```
(* utiliza el vector b como auxiliar para realizar la mezcla *)
   VAR mitad:CARDINAL;
BEGIN
   IF prim<ult THEN
      IF (ult-prim)<M THEN Seleccion(a,prim,ult)
      ELSE
         mitad:=(prim+ult)DIV 2;
         Mezcla2(a,b,prim,mitad);
         Mezcla2(a,b,mitad+1,ult);
         Combinar(a,b,prim,mitad,mitad+1,ult)
      END
   END
END Mezcla2;
```

Solución al Problema 2.5. (☺)

Tras el estudio que se hizo en el apartado 2.6 sobre la complejidad del método, podemos ya seleccionar dos casos en donde el algoritmo se va a comportar de forma especial:

– Pensemos lo que ocurre cuando todos los elementos del vector a ordenar son iguales. En este caso la función que "empuja" es O(1), con lo cual la complejidad del algoritmo es:

$$(n/2)O(1) + nO(1)O(1) \in O(n)$$

– Cuando los elementos del vector son todos distintos y están ya ordenados en forma creciente, el procedimiento que construye el montículo es de complejidad $O(n)$, y además nos deja un montículo en donde en cada iteración se va a tomar como nueva raíz el mínimo del vector, lo que implica que ésta habrá de ser empujada a lo largo de todo el montículo. Esto hace que su complejidad sea:

$$O(n) + O(n\log n) \in O(n\log n)$$

En resumen, el caso mejor para el algoritmo de ordenación por Montículos ocurre cuando los elementos a ordenar son todos iguales, y el peor cuando son todos distintos y además el vector está ya ordenado.

Es interesante que el lector compare estos casos con los casos desfavorables para Quicksort, estudiados en el problema 2.8.

Solución al Problema 2.6. (☺)

El código de los procedimientos Mezcla y Quicksort en su versión recursiva ya ha sido visto en las secciones correspondientes del presente capítulo. Veamos por tanto la versión iterativa de ambos métodos.

Procedimiento Mezcla Iterativo

Comenzaremos con el procedimiento de ordenación por Mezcla, que admite una versión iterativa que no se basa directamente en una eliminación de la recursión.

La idea es ir dando pasadas por el vector haciendo mezclas. En la primera pasada se mezclan los elementos adyacentes para formar subvectores ordenados de tamaño dos. En la siguiente pasada se mezclan los subvectores adyacentes de tamaño dos para formar subvectores ordenados de tamaño cuatro. Así se continúa hasta obtener un sólo vector ordenado de tamaño *n*. En general van a hacer falta log*n* pasadas para ordenar un vector de *n* elementos. Esto da lugar a la siguiente implementación del algoritmo:

```
PROCEDURE Mezcla_it(VAR a:vector;prim,ult:CARDINAL);
   VAR l,p1,p2,u1,u2:CARDINAL;
BEGIN
   l:=1; (* l es el num. elementos de la mezcla en cada paso *)
   WHILE l<(ult-prim+1) DO
      p1:=prim;
      WHILE p1<ult DO
         u1:=p1+l-1;
         p2:=u1+1;
         u2:=p2+l-1;
         IF p2<=ult THEN
            IF u2>ult THEN u2:=ult END;
            Combinar(a,b,p1,u1,p2,u2)
         END;
         p1:=u2+1
      END;
      l:=2*l
   END
END Mezcla_it;
```

El procedimiento *Combinar* es el mismo que fue implementado para el método *Mezcla*.

Procedimiento Quicksort Iterativo

La versión iterativa de Quicksort se basa en el mecanismo de eliminación de la recursión mediante el uso de una pila que va a contener el trabajo pendiente en cada momento.

Como puede observarse en el algoritmo que presentamos a continuación, vamos a ir dividiendo el vector en dos subvectores (con la función *Pivote*) e insertando en una pila que hemos creado al efecto las posiciones de comienzo y fin de un subvector que más tarde ordenaremos.

Con el otro subvector, en vez de almacenarlo en la pila, lo iremos dividiendo y guardando sus mitades en la pila. Por tratarse de un caso de recursión de cola, podremos eliminar la llamada recursiva mediante un bucle en donde en cada iteración calculamos los valores de las variables para la siguiente.

Una vez acabamos con una mitad, extraemos otro subvector de la pila y repetimos el proceso con él, y así hasta que vaciemos la pila, lo que indicará que ya

hemos ordenado el vector (hemos ordenado cada uno de los "trozos" en los que lo habíamos dividido).

```
FROM PILAS IMPORT pila,Crear,Destruir,Insertar,Extraer,Esvacia;
(* usamos pilas de CARDINAL para almacenar posiciones *)

PROCEDURE Quicksort_it(VAR a:vector;prim,ult:CARDINAL);
   VAR i:CARDINAL; p:pila;
BEGIN
   Crear(p);
   LOOP
      WHILE ult>prim DO
         i:=Pivote(a,a[prim],prim,ult);
         Insertar(p,prim);
         Insertar(p,i-1);
         prim:=i+1
      END;
      IF Esvacia(p) THEN
         Destruir(p);
         RETURN
      END;
      ult:=Extraer(p);
      prim:=Extraer(p)
   END
END Quicksort_it;
```

Obsérvese que es la función *Pivote* la encargada de mover los elementos del vector, y por eso no aparece ninguna instrucción de intercambio en el código que presentamos.

Este algoritmo procesa los mismos subvectores que su versión recursiva, aunque en orden distinto. Además, esta versión iterativa supone un incremento en la eficiencia y admite alguna variante o mejora interesante:

- Por un lado, si en vez de meter en la pila el primero de los dos subvectores metiéramos el de mayor tamaño, conseguiríamos que el tamaño de la pila no fuera nunca mayor que $\log n$, puesto que cada entrada en la pila después de la primera debe representar un subvector con menos de la mitad de los elementos de la entrada anterior. Ésta es una mejora interesante con respecto a la versión recursiva de Quicksort, pues en el peor de sus casos la pila de recursión usada puede alcanzar las n entradas (por ejemplo, cuando el vector está ordenado inicialmente). Así se minimiza el riesgo que siempre conlleva el desbordar la memoria existente por un crecimiento desmesurado de la pila de recursión.
- Por otro lado, también podemos mejorar esta versión iterativa para tratar el caso trivial que se obtiene cuando partimos un subvector en dos subvectores de tamaño 1. En este caso se inserta una entrada en la pila para ser extraída inmediatamente y después descartada. Es muy fácil cambiar el algoritmo para

tener en cuenta este hecho, pero quizá sea también mejor tratar los subvectores de tamaño pequeño por separado (como se muestra en el problema 2.4).

Solución al Problema 2.7. (😐)

Respecto a la secuencia de incrementos, lo que parece deseable es que los números que la componen sean coprimos entre sí (dos números se dicen coprimos si su máximo común divisor es 1), pues si no pueden producirse iteraciones en donde no haya intercambios de elementos aunque sí un gran número de comparaciones. Con esto en mente es fácil encontrar el ejemplo pedido. Suponiendo el vector:

[12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Después de las sucesivas pasadas obtenemos:

		Número de Comparaciones	Número de Intercambios
h=8	[4, 3, 2, 1, 8, 7, 6, 5, 12, 11, 10, 9]	4	4
h=4	[4, 3, 2, 1, 8, 7, 6, 5, 12, 11, 10, 9]	8	0
h=2	[2, 1, 4, 3, 6, 5, 8, 7, 10, 9, 12, 11]	10	4
h=1	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]	11	6

Como puede verse en este ejemplo, el problema es el número de comparaciones frente al número de intercambios. El primero es fijo para esta secuencia (del orden de n^2) pero sin embargo no consigue hacer disminuir el número de intercambios de elementos. Puede verse en el ejemplo cómo los números pares e impares se ordenan por separado, pero ninguno en su posición definitiva hasta el último paso (*h*=1), en el que lo hacen todos los elementos del vector. Este inconveniente hace que el algoritmo pierda en este caso su competitividad al compararlo con algoritmos aún más sencillos de codificar y depurar como son el de Inserción o Selección, y no digamos si la comparación se realiza frente a procedimientos de complejidad $n\log n$ como pueden ser Quicksort o Mezcla.

Solución al Problema 2.8. (😐)

En la implementación que hemos expuesto del método Quicksort podemos ver que si los elementos son todos iguales la complejidad del algoritmo es de orden cuadrático. En efecto, la partición por el pivote consume un tiempo del orden de $O(n)$, pero el pivote resulta ser siempre el primer elemento, con lo cual una de las dos llamadas a Quicksort tiene tamaño cero y la otra tamaño $n-1$. Este desequilibrio hace que la complejidad del algoritmo resultante sea de orden $O(n^2)$.

- La primera idea para eliminar ese efecto indeseable es la de comprobar tal condición antes de comenzar a ejecutar el algoritmo, lo que da lugar al siguiente procedimiento:

```
PROCEDURE Quicksort3(VAR a:vector;prim,ult:CARDINAL);
```

```
   VAR l:CARDINAL;
BEGIN
   IF prim<ult THEN
      IF SonTodosIguales(a,prim,ult) THEN RETURN END;
      l:=Pivote(a,a[prim],prim,ult);
      Quicksort3(a,prim,l-1);
      Quicksort3(a,l+1,ult)
   END
END Quicksort3;
```

Naturalmente, la función *SonTodosIguales* devuelve *TRUE* si todos los elementos del subvector *a*[*prim..ult*] son iguales, *FALSE* en otro caso, y es de complejidad O(n).

Esta modificación es simple y fácil de implementar, y mantiene en O($n$log$n$) la complejidad del algoritmo para su caso medio, eliminando el caso conflictivo. Sin embargo, la constante que se obtiene en su tiempo de ejecución hace que la modificación no sea ya rentable frente a otros métodos como Montículos o Mezcla. ¿Existe otra posible solución?

- Una idea alternativa para eliminar este caso conflictivo es menos intuitiva, pero mucho más efectiva. Se basa en modificar la función *Pivote*, de forma que ahora divida el vector inicial en tres partes. Al finalizar la función, los elementos del vector *a* habrán sido permutados de forma que todos los elementos menores que el pivote *p* estén a su izquierda, los elementos iguales a *p* se encuentren todos juntos en el centro, y los elementos mayores que *p* a su derecha. La función devuelve dos posiciones, que son las que marcan las tres partes en las que hemos dividido el vector *a*:

```
PROCEDURE Pivote2(VAR a:vector; p:INTEGER; prim,ult:CARDINAL;
                  VAR k,l:CARDINAL);
(* permuta los elementos de a[prim..ult] y devuelve dos
   posiciones k,l tales que prim-1<=k<l<=ult+1, a[i]<p si
   prim<=i<=k, a[i]=p si k<i<l, y a[i]>p si l<=i<=ult, donde p es
   el pivote y aparece como argumento *)
   VAR m:CARDINAL;
BEGIN
   k:=prim; l:=ult+1;
   (* primero buscamos l *)
   REPEAT INC(k) UNTIL (a[k]>p) OR (k>=ult);
   REPEAT DEC(l) UNTIL (a[l]<=p);
   WHILE k<l DO
      Intercambia(a,k,l);
      REPEAT INC(k) UNTIL (a[k]>p);
      REPEAT DEC(l) UNTIL (a[l]<=p)
   END;
   Intercambia(a,prim,l);
```

```
    INC(l); (* ya tenemos l *)
    (* ahora buscamos el valor de k *)
    k:=prim-1;
    m:=l;
    REPEAT INC(k) UNTIL (a[k]=p) OR (k>=l);
    REPEAT DEC(m) UNTIL (a[m]<>p) OR (m<prim);
    WHILE k<m DO
       Intercambia(a,k,m);
       REPEAT INC(k) UNTIL (a[k]=p);
       REPEAT DEC(m) UNTIL (a[m]<>p);
    END;
END Pivote2;
```

Una vez disponemos de ese procedimiento, cuya complejidad es lineal, sólo queda modificar ligeramente el código de *Quicksort*:

```
PROCEDURE Quicksort4(VAR a:vector;prim,ult:CARDINAL);
   VAR k,l:CARDINAL;
BEGIN
   IF prim<ult THEN
      Pivote2(a,a[prim],prim,ult,k,l);
      Quicksort4(a,prim,k);
      Quicksort4(a,l,ult)
   END
END Quicksort4;
```

Con esta modificación Quicksort ofrece una complejidad lineal cuando los elementos del vector a ordenar son todos iguales (en este caso se tiene que $k = prim$ y $l = ult$ a la salida del procedimiento *Pivote2*). Sin embargo, ¿cuál ha sido el precio? Para el caso medio hemos "empeorado" el tiempo de ejecución del algoritmo; aunque sigue siendo de complejidad O($n$log$n$), las constantes que acompañan a los coeficientes de la función de su tiempo de ejecución lo han convertido en un algoritmo más ineficiente que los métodos de ordenación por Montículos y Mezcla en el caso medio.

Ya hemos visto cómo solucionar uno de los casos desfavorables para Quicksort, que es cuando los elementos del vector a ordenar son todos iguales. Sin embargo, la implementación que hemos realizado de Quicksort posee otros casos desfavorables: cuando el vector a ordenar está ya ordenado en orden creciente o decreciente.

En ambos casos se repite lo que ocurría cuando los elementos eran iguales. Se produce un desequilibrio en los subproblemas que resultan de la división del vector, ya que de las dos llamadas recursivas a Quicksort una es invocada con 0 elementos y la otra con n–1. Esto lleva a que la complejidad del algoritmo en este caso sea también del orden de O(n^2).

Sin embargo, este problema admite una solución más fácil que el que hemos visto anteriormente. Basta coger como pivote la mediana del vector, en vez de su

primer elemento. Así las llamadas recursivas estarán siempre equilibradas y si la búsqueda de la mediana se hace en tiempo lineal la complejidad del algoritmo se mantendrá del orden de O(*n*log*n*). Para ver cómo puede determinarse la mediana de un vector, consúltese el problema 2.15.

Solución al Problema 2.9. (☺)

El procedimiento de ordenación por Residuos para números naturales utiliza una serie de colas auxiliares donde irá clasificando los elementos del vector. Se definen tantas colas como números distintos posea la base de representación usada (que llamaremos *B*). Por ejemplo, si decidimos utilizar el método para una representación decimal (*B* = 10), dispondremos de 10 colas (numeradas del 0 al 9); si la representación es binaria dispondremos de 2 colas (0 y 1).

Este método funciona mediante un bucle en el cual en cada iteración considera un dígito distinto de cada uno de los elementos del vector, de derecha a izquierda, insertando el elemento en cuestión en la cola correspondiente al valor de su dígito. Por ejemplo, para una representación binaria, en la primera iteración insertará los elementos en cada una de las dos colas de acuerdo a su dígito menos significativo. En la segunda respecto a su segundo dígito menos significativo, y así sucesivamente.

Tras cada iteración, una vez los elementos hayan sido clasificados utilizando las colas, el método extrae los elementos de las colas en orden, volcándolos de nuevo en el vector. Por tanto, en cada iteración los elementos están en orden respecto al dígito correspondiente a tal iteración. Como utilizamos estructuras de colas (con estrategia de acceso FIFO) se respeta el orden relativo de los elementos de una iteración a otra, lo que hace que el método consiga ordenar finalmente el vector. Naturalmente, el número de iteraciones a realizar va a coincidir con el logaritmo en base *B* del mayor elemento del vector.

Una posible implementación de este algoritmo para una base *B* general es el que sigue, donde *MAXB* es una constante que indica el valor máximo para *B*:

```
TYPE colaitem = RECORD elems:vector; cont:CARDINAL END;

PROCEDURE Residuos(VAR a:vector;B,prim,ult:CARDINAL);
   VAR  i,j,k,h,digito:CARDINAL;
        iter:INTEGER;
        sigo:BOOLEAN;
        colas: ARRAY [0..MAXB-1] OF colaitem;
BEGIN
   iter:=1;                (* iter va acumulando B^1,B^2,B^3,... *)
   REPEAT                  (* para cada uno de los digitos *)
      FOR i:=0 TO B-1 DO   (* inicializamos las colas *)
         colas[i].cont:=1  (* primera posicion libre *)
      END;
      (* clasificacion de los elementos en las colas *)
      sigo:=FALSE;(* indica si quedan numeros con mas digitos *)
      FOR i:=prim TO ult DO
         sigo:=((a[i] DIV iter)>=INTEGER(B)) OR sigo;
```

```
         digito:=(a[i] DIV iter) MOD INTEGER(B); (* num cola *)
         colas[digito].elems[colas[digito].cont]:=a[i];
         INC(colas[digito].cont)    (* inserta a[i] en su cola *)
      END;
      iter:=iter*INTEGER(B);
      j:=prim; (*ahora volcamos las colas sobre el vector *)
      FOR i:=0 TO B-1 DO
         h:=colas[i].cont-1;   (* num de elementos en esa cola *)
         FOR k:=1 TO h DO a[j]:=colas[i].elems[k]; INC(j) END
      END
   UNTIL NOT sigo;
END Residuos;
```

Como puede observarse, el método realiza un ciclo principal para cada uno de los dígitos, en donde existen tres bucles por cada iteración: primero se inicializan las colas, después se clasifican los elementos en las colas y por último se vuelcan ordenadamente las colas en el vector.

Estudiaremos a continuación la complejidad (temporal) de tal algoritmo. Para ello, si llamamos x al mayor de los elementos del vector a, el número de operaciones elementales que se efectúan en una llamada a *Residuos(a,B,*1*,n)* es:

$$T_B(n) = 1+(5B+2+1+2+(2+5+5+7+3)n+2+1+2+8n+8B)\log_B x =$$
$$= 1+(10+30n+13B)\log_B x.$$

En esta ecuación, la cantidad $\log_B x$ indica el número de dígitos del mayor de los elementos del vector, cuando éste se expresa en base B. Para tratar de cuantificar el valor de la ecuación, podemos acotar el valor de $\log_B x$ por la cantidad $\log_B C$, donde C es la constante que indica el máximo número entero que puede representarse en nuestra máquina. De esta forma, para un ordenador cuya palabra sea de 16 bits tenemos que $\log_2 C = 15$ y $\log_{10} C = 5$, por lo que:

$$T_2(n) \le 1+(36+30n)\log_2 C = 1+(36+30n)\cdot 15 = 450n+541.$$

$$T_{10}(n) \le 1+(140+30n)\log_{10} C = 1+(140+30n)\cdot 5 = 150n+701.$$

Ambos tiempos de ejecución son lineales, y puede observarse que el segundo es menor que el primero. Pero, ¿cuál es el precio que estamos pagando en ambos casos?

La respuesta a esta pregunta viene dada por la complejidad espacial de los algoritmos. Es cierto que ambos consiguen ordenar un vector muy eficientemente, pero también tienen un orden de complejidad espacial lineal. De hecho, el primer algoritmo utiliza del orden de $2n$ elementos auxiliares mientras que el segundo utiliza del orden de $10n$.

Sería posible mejorar esta complejidad espacial mediante el uso de estructuras dinámicas para implementar las colas. Sin embargo este cambio trae consigo un aumento de la complejidad temporal del algoritmo, pues el acceso a estas estructuras es más lento.

Solución al Problema 2.10. (☺)

Para dar respuesta a este problema, es importante hacer notar que la estabilidad de un método depende en gran medida de su implementación. Un ejemplo claro es el método de Selección, en donde la forma de encontrar el elemento mínimo de entre los que quedan por ordenar es clave a la hora de la estabilidad del método. Por eso nos basaremos en las implementaciones de los métodos que se presentan en este capítulo.

Los métodos estables son:

Inserción: Cada elemento se coloca en su lugar, respetando el orden de llegada y por tanto su orden parcial.

Burbuja: Este método es análogo a Inserción y como él, respeta el orden relativo de los elementos.

Mezcla: Como en este método los elementos sólo se intercambian en el procedimiento *Combinar*, es suficiente probar que las mezclas son estables. Pero esto es cierto puesto que la forma de mezclar los subvectores respeta los órdenes relativos de los elementos que los componen.

Los métodos no estables son:

Selección: Aunque la búsqueda del elemento mínimo se hace en el mismo sentido en el que se está ordenando el vector y se escoge siempre el primero de ellos en caso de haber más de uno, este método no resulta ser estable. Por ejemplo, para el vector [2_A, 2_B, 1, 3] (indicamos el orden relativo original de cada elemento mediante el subíndice) tras la primera vuelta, en donde se intercambian los elementos 2_A y 1, obtenemos el vector [1, 2_B, 2_A, 3] que ya no se altera en ninguna pasada.

Incrementos: Este método puede ser visto como una sucesión de aplicaciones del método de Inserción sobre cada uno de los *h*-vectores. Para encontrar un ejemplo que demuestre que el método no es estable, basta coger dos elementos iguales que correspondan a dos *h*-vectores distintos para un valor dado de *h*. Al ordenarse estos dos *h*-vectores por separado, no se respeta el orden parcial de los elementos. Por ejemplo, para una secuencia de incrementos (*h*) que acabe en 4 y 1, podemos tomar el vector [2_A, 2_B, 1_A, 1_B, 1_C]. Aplicando el algoritmo *Incrementos* implementado en el apartado 2.8 obtenemos el vector [1_C, 1_A, 1_B, 2_B, 2_A] que, como puede comprobarse, no conserva el orden relativo de los elementos.

Quicksort: Los elementos se intercambian en este método sólo en la función *Pivote*, pero es fácil ver que ésta no respeta su orden relativo, por la forma en que va intercambiando los elementos al hacerlos "saltar" de

un lado del pivote al otro. De esta forma, al ordenar el vector [2_A, 2_B, 1_A, 1_B, 3] obtiene como resultado [1_A, 1_B, 2_B, 2_A, 3].

Montículos: Este método no es estable por la filosofía que tiene de intercambiar el primer elemento del montículo (que es el mayor) con el último. Así, el vector [2_A, 2_B, 1_A, 1_B] que ordenado como [1_B, 1_A, 2_B, 2_A] aplicando el algoritmo implementado en el apartado 2.6.

Ladrillos: Este método no es estable por la forma en la que se escoge el máximo en cada iteración. La función *PosMaximo* elige el elemento máximo con el menor subíndice, y el algoritmo *Ladrillos* lo coloca al final: realmente este algoritmo está invirtiendo el orden relativo inicial de los elementos. Como ejemplo, el vector [2_A, 2_B, 1_A, 1_B, 3] queda ordenado como [1_B, 1_A, 2_B, 2_A, 3].

Sacudidas: Este método se encuentra en la misma circunstancia que Ladrillos, por el mismo motivo. El resultado que obtiene tras ordenar el vector [2_A, 2_B, 2_C, 1_A, 1_B] es [1_B, 1_A, 2_C, 2_B, 2_A].

Por último, para algunos de los métodos no estables es fácil sugerir modificaciones que los conviertan en estables, mientras que para otros no es posible debido a la filosofía general de funcionamiento de cada uno de ellos. Por ejemplo, en los métodos de ordenación por Incrementos y Montículos no se puede mantener el orden relativo de los elementos sin alterar el diseño de los métodos.

Sin embargo, sí es fácil conseguir versiones estables de los demás. Por ejemplo, para el método de Selección basta con ir copiando el vector a otro, de forma que en cada pasada se copie el elemento mínimo:

```
PROCEDURE Seleccion3(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
   VAR i,j:CARDINAL; b:vector;
BEGIN
   FOR i:=prim TO ult DO b[i]:=a[i] END; (* copiamos a en b *)
   FOR i:=prim TO ult DO
      j:=PosMinimo(b,prim,ult);
      a[i]:=b[j]
      b[j]:=MAX(INTEGER); (* para no tenerlo en cuenta mas *)
   END;
END Seleccion3;
```

Para *Ladrillos* y *Sacudidas*, no sólo basta con que utilicen una función de selección del máximo que escoja el elemento mayor que aparezca en última posición, sino que también tendrían que incorporar un cambio similar al sugerido para el método de Selección.

Respecto a *Quicksort*, la única modificación a realizar es en la función *Pivote*, de forma que respete el orden parcial de los elementos cuando los hace "saltar" de un lado del pivote a otro.

Solución al Problema 2.11. (👓)

El procedimiento de Inserción modificado puede ser implementado como sigue:

```
PROCEDURE Insercion2(VAR a:vector;prim,ult:CARDINAL);
   VAR i,j,k:CARDINAL; x:INTEGER;
BEGIN
   FOR i:=prim+1 TO ult DO
      x:=a[i]; k:=Posicion(a,prim,i-1,x);
      FOR j:=i-1 TO k+1 BY -1 DO
         a[j+1]:=a[j]
      END;
      a[k]:=x
   END
END Insercion2;
```

Nos apoyamos en una función que calcula la posición en donde hemos de insertar un nuevo elemento dentro de un subvector previamente ordenado, utilizando búsqueda binaria:

```
PROCEDURE
Posicion(a:vector;prim,ult:CARDINAL;x:INTEGER):CARDINAL;
   VAR mitad:CARDINAL;
BEGIN
   WHILE (prim<=ult) DO
      mitad:=(prim+ult) DIV 2;
      IF x=a[mitad] THEN RETURN mitad
      ELSIF x<a[mitad] THEN ult:=mitad-1
      ELSE prim:=mitad+1
      END
   END;
   RETURN mitad
END Posicion;
```

Para calcular la complejidad del nuevo procedimiento de Inserción, necesitamos conocer primero el tiempo de ejecución de la función *Posicion*. Pero éste es conocido, pues es igual al de la función *BuscBIt* del problema 1.16 (Búsqueda binaria implementada de forma iterativa). Con esto, la complejidad de *Insercion2* es como sigue:

– Para el estudio del caso mejor, tenemos que éste puede ocurrir cuando (*a*) se da el caso mejor de la función *Posicion*, con lo que el bucle siguiente se efectúa para la mitad de los elementos considerados. O bien cuando (*b*) el bucle se efectúa una vez por ser $k = i - 1$; que es también el peor caso de *Posicion*. Calculando el tiempo de ejecución de cada una de las opciones ($T_a(n)$ y $T_b(n)$):

$$T_a(n) = \left(\sum_{i=1}^{n-1} (11 + 9 + 6(i-1)/2) \right) + 2 = \frac{3}{2}n^2 + \frac{31}{2}n - 15.$$

$$T_b(n) = \left(\sum_{i=1}^{n-1} (11 + 10\log i + 4 + 0) \right) + 2 = 15n - 13 + 10\sum_{i=1}^{n-1} \log i.$$

Como $T_a(n) > T_b(n)$ para valores grandes de n, el mejor de los casos de *Insercion2* corresponde a la opción (*b*), que se produce cuando el vector está previamente ordenado.

– El caso peor ocurre cuando se dan simultáneamente los casos peores de la función *Posicion* y del bucle *WHILE* (y esto ocurre cuando el vector está ordenado en forma inversa a como queremos ordenarlo). Llamando (*p*) a esta opción obtenemos:

$$T_p(n) = \left(\sum_{i=1}^{n-1} (11 + 10\log i + 4 + 6(i-1)) \right) + 2 = 3n^2 + 6n - 7 + 10\sum_{i=1}^{n-1} \log i.$$

– Por último, el caso medio ocurre cuando se da el caso medio de la función *Posicion* y el bucle *WHILE* se ejecuta, en media, $i/2$ veces:

$$T_{1/2}(n) = \left(\sum_{i=1}^{n-1} \left(11 + 8\frac{i\log i - i + 1}{i+1} + 6 + 6(i-1)/2 \right) \right) + 2.$$

Respecto a los órdenes de complejidad de tales funciones, todos son cuadráticos menos en la opción (*b*), de complejidad O($n\log n$) por ser de este orden la expresión $\sum_{i=1}^{n-1} \log i$.

Para valorar los dos algoritmos necesitamos comparar sus tiempos de ejecución siempre que estos ocurran bajo las mismas circunstancias. Pero así sucede en este algoritmo: el caso peor de ambos métodos ocurre cuando el vector está ordenado en orden inverso al deseado, el caso medio cuando cualquier permutación del vector es inicialmente equiprobable, y el caso mejor cuando el vector a ordenar está ya ordenado. Podemos comparar entonces sus tiempos de ejecución y obtenemos:

- En el caso mejor $T_{Insercion}(n) < T_{Insercion2}(n)$, siendo incluso el segundo de un orden de complejidad mayor al primero (n frente a $n\log n$).
- En el caso peor $T_{Insercion}(n) > T_{Insercion2}(n)$ para $n > 15$, aunque ambos algoritmos son de complejidad cuadrática.
- En el caso medio $T_{Insercion}(n) > T_{Insercion2}(n)$ para $n > 23$, aunque ambos algoritmos son de complejidad cuadrática.

Como puede observarse, el algoritmo modificado (*Insercion2*) es más eficiente (respecto a su tiempo de ejecución) que el algoritmo inicial para los casos peor y medio. Esto era de esperar porque, aunque no se rebaja el número de intercambios que dejan al nuevo elemento en su posición final, sí se consigue disminuir el número de comparaciones necesario para buscar tal posición.

Sin embargo, en el caso mejor estamos introduciendo una serie de comparaciones adicionales por la forma en la que buscamos. Esto hace aumentar la complejidad del algoritmo en un orden de magnitud, lo cual no es despreciable.

Resumiendo, con esta modificación conseguimos una mejora en la mayoría de los casos respecto al tiempo de ejecución. Pero obsérvese que esto siempre lleva asociado un precio. La simplicidad del algoritmo original se ve comprometida en esta nueva versión, lo que suele conllevar problemas de depuración, verificación y mantenimiento del nuevo código. Como siempre, dejamos en manos del usuario del algoritmo la decisión a tomar, pues ésta va a depender mucho de los factores particulares de su sistema o aplicación.

Solución al Problema 2.12. (☺)

El procedimiento *Rastreo* puede ser implementado como sigue:

```
PROCEDURE Rastreo (VAR a:vector;prim,ult:CARDINAL);
   VAR i:CARDINAL;
BEGIN
   IF prim>=ult THEN RETURN END;
   i:=prim;
   LOOP
      WHILE a[i]<=a[i+1] DO
         INC(i); IF i=ult THEN RETURN END
      END;
      Intercambia(a,i,i+1);
      WHILE (i>prim) AND (a[i-1]>a[i]) DO
         Intercambia(a,i,i-1);
         DEC(i)
      END
   END
END Rastreo;
```

En cuanto a su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Rastreo(a,*1*,n)*.

– El caso mejor ocurre cuando el algoritmo nunca tiene que ir "hacia atrás", limitándose a ejecutar sólo el primer ciclo *WHILE*, resultando:

$$T(n) = 1+1+\left(\sum_{i=1}^{n-1}(1+1+6)\right)+1 = 8n-5 .$$

Obsérvese que este caso mejor va a coincidir cuando el vector está inicialmente ordenado y por tanto sólo se efectúa el primer bucle.

– En el caso peor los dos bucles *WHILE* se ejecutan siempre en su totalidad. Así, obtenemos:

$$T(n) = 1+1+\sum_{i=1}^{n-1}\left(\left(\sum_{k=1}^{i-1}(1+1+6)\right)+6+7+2+\left(\sum_{k=1}^{i-1}(1+7+1+1+6)\right)+6\right)$$

$$= 12n^2 - 15n + 5\,.$$

Esta situación sucede cuando el vector está inicialmente ordenado en forma inversa a como queremos ordenarlo.

– En el caso medio, cualquier ordenación de los elementos es igualmente probable, y por tanto el número de veces que se repite cada uno de los bucles *WHILE* es:

$$T(n) = 1+1+\sum_{i=1}^{n-1}\left(\frac{1}{2}\left(\sum_{k=1}^{i-1}(1+1+6)\right)+6+7+2+\frac{1}{2}\left(\sum_{k=1}^{i-1}(1+7+1+1+6)\right)+6\right)$$

$$=6n^2 + 3n - 7\,.$$

Solución al Problema 2.13. (☹)

De forma análoga al procedimiento *Mezcla* expuesto al principio del capítulo, el procedimiento pedido puede ser implementado como sigue:

```
PROCEDURE MezclaTres(VAR a,b:vector;prim,ult:CARDINAL);
(* utiliza el vector b como auxiliar para realizar la mezcla *)
   VAR terc1,terc2:CARDINAL;
BEGIN
   IF ult<=prim THEN RETURN END;
   IF ult-prim>=2 THEN
      (* primero divide el vector en tres subvectores:
         a[prim..terc1], a[terc1+1..terc2] y a[terc2+1..ult] *)
      terc1:=prim-1+(ult-prim+1)DIV 3;
      terc2:=terc1+(ult-prim+2)DIV 3;
      MezclaTres(a,b,prim,terc1);
      MezclaTres(a,b,terc1+1,terc2);
      MezclaTres(a,b,terc2+1,ult);
      (* y luego los mezcla *)
      CombinarTres(a,b,prim,terc1,terc1+1,terc2,terc2+1,ult)
   ELSE  (* caso base *)
      IF a[prim]>a[ult] THEN Intercambia(a,prim,ult) END
   END
END MezclaTres;
```

En cuanto al procedimiento que realiza la mezcla, es una versión análoga a la del algoritmo original pero esta vez combinando tres subvectores consecutivos y ya ordenados.

Primero vuelca los elementos a ordenar en el vector auxiliar y luego, utilizando un índice para cada subvector, va escogiendo el menor de los elementos apuntados por esos índices e incrementando el índice correspondiente hasta alcanzar el final.

Nótese que el algoritmo utiliza el hecho de que los subvectores están ya ordenados y que además son consecutivos.

Su implementación es como sigue:

```
PROCEDURE CombinarTres(VAR a,b:vector;
                       p1,u1,p2,u2,p3,u3:CARDINAL);
(* mezcla ordenadamente los subvectores a[p1..u1], a[p2..u2] y
   a[p3..u3] suponiendo que estos estan ya ordenados y que son
   consecutivos (es decir, p2=u1+1 y p3=u2+1), y utilizando el
   vector b como auxiliar. *)
   VAR i1,i2,i3,k:CARDINAL;
BEGIN
   FOR k:=p1 TO u3 DO b[k]:=a[k] END;
   i1:=p1;i2:=p2; i3:=p3;   (* cada indice se encarga de un
subvector *)
   FOR k:=p1 TO u3 DO
      CASE NumMin(b[i1],b[i2],b[i3]) OF
      |1:a[k]:=b[i1];
         IF i1<u1 THEN INC(i1) ELSE b[i1]:=MAX(INTEGER) END
      |2: a[k]:=b[i2];
         IF i2<u2 THEN INC(i2) ELSE b[i2]:=MAX(INTEGER) END
      |3: a[k]:=b[i3];
         IF i3<u3 THEN INC(i3) ELSE b[i3]:=MAX(INTEGER) END
      END
   END
END CombinarTres;
```

Por otro lado, para escoger cuál de los elementos es menor utiliza una función auxiliar que calcula el orden relativo del mínimo de tres elementos:

```
PROCEDURE NumMin(a,b,c:INTEGER):CARDINAL;
BEGIN
   IF (a<=b) AND (a<=c) THEN RETURN 1 END; (* a es el menor *)
   IF (b<=a) AND (b<=c) THEN RETURN 2 END; (* b es el menor *)
   RETURN 3; (* c es el menor *)
END NumMin;
```

Para calcular la complejidad de este algoritmo, determinaremos primero su tiempo de ejecución en función del número de operaciones elementales que realiza dependiendo del tamaño de la entrada (número de elementos a ordenar).

Siguiendo el mismo método que hemos utilizado en los problemas del primer capítulo, se llega a que el tiempo de ejecución de *MezclaTres(a,b,*1*,n)* puede expresarse mediante una ecuación en recurrencia:

$$T_3(n) = 3T_3(n/3) + 21n + 29$$

con la condición inicial $T_3(1) = 2$. Ésta es una ecuación en recurrencia no homogénea cuya ecuación característica es $(x-3)^2(x-1) = 0$, lo que permite expresar $T_3(n)$ como:

$$T_3(n) = c_1n + c_2n\log_3 n + c_3.$$

El cálculo de las constantes puede hacerse a partir de la condición inicial, lo que nos lleva a la expresión final:

$$T_3(n) = 21n\log_3 n + (33/2)n - (29/2) \in \Theta(n\log n).$$

Comparemos a continuación este procedimiento con el de Mezcla clásico. Para eso necesitamos calcular su tiempo de ejecución, y nos apoyaremos en la implementación que hemos dado al principio de este capítulo.

El tiempo de ejecución de *Mezcla(a,b,*1*,n)* ya lo calculamos cuando expusimos tal método, obteniendo la siguiente expresión:

$$T_2(n) = 16n\log n + 18n - 17 \in \Theta(n\log n).$$

Como podemos observar, ambos métodos son del mismo orden de complejidad. Ahora bien, nos planteamos si uno de ellos es mejor que el otro, en qué casos y cuánto mejor.

Para responder a estas preguntas tendremos que comparar las funciones que definen los tiempos de ejecución de ambos algoritmos. Para eso definimos:

$$T_d(n) = T_2(n) - T_3(n).$$

Puede comprobarse que $T_d(n) > 0$ para todo $n > 2$. Esto implica que el algoritmo *MezclaTres* se comporta mejor que el de Mezcla original, aunque siempre teniendo en cuenta que ambos son del mismo orden de complejidad.

$T_d(n)$ nos ofrece una medida absoluta del grado de mejora que supone un método frente a otro. Por otro lado, la expresión

$$\lim_{n\to\infty} \frac{T_2(n)}{T_3(n)} = 1.20759$$

nos indica que, para valores grandes de n, el método de Mezcla clásico es hasta un 20% peor que el segundo algoritmo.

Sin embargo, ¿cuál es el precio que hemos pagado para conseguir esta mejora? Sin lugar a dudas este precio se refleja en el código resultante, más complicado, difícil de diseñar y mantener. Este aspecto, como ya hemos mencionado anteriormente, debe tenerse siempre en cuenta y explica la razón por la que se utiliza normalmente el procedimiento que parte en dos a pesar de saber que el que parte en tres es un poco más eficiente (en tiempo de ejecución).

Tras el resultado obtenido en este problema podemos plantearnos qué ocurriría si partiésemos el vector en cuatro partes. ¿Volveríamos a obtener una mejora? ¿Y en cinco partes? ¿Existe un número óptimo de partes en las que dividir el vector? Aunque no entraremos en detalle para responder estas cuestiones, sí queremos dar una idea intuitiva de lo que ocurre en estos casos.

En primer lugar, hemos visto que el tiempo de ejecución del método clásico es de la forma $T_2(n) = an\log_2 n + b$, y el de *MezclaTres* puede expresarse como:

$$T_3(n) = cn\log_3 n + d = (c/\log 3)n\log_2 n + d,$$

donde a, b, c y d son constantes. En general, el tiempo de ejecución del método basado en dividir el vector en k partes va a ser

$$T_k(n) = skn\log_k n + t = s(k/\log k)n\log_2 n + d,$$

siendo s y t constantes y en donde vamos a poder conseguir que el valor de s sea muy similar al de c por la estructura del algoritmo. La razón por la que se introduce la k como constante multiplicativa es debido a la función que ha de calcular el mínimo de los elementos del vector auxiliar b. Para el caso de Mezcla clásico es el mínimo de dos elementos; para *MezclaTres* han de considerarse tres elementos, y para "*MezclaK*" es necesario encontrar el mínimo de k elementos no necesariamente ordenados, procedimiento de orden de complejidad lineal –O(k)–.

Esto nos lleva a que todos los métodos van a ser del mismo orden de complejidad. Sin embargo, las funciones $T_k(n)$ son cada vez mayores conforme k crece, puesto que

$$\lim_{k\to\infty} \frac{k}{\log_2 k} = \infty .$$

En resumidas cuentas, aunque inicialmente $T_2(n)$ fuera mayor que $T_3(n)$ esto no va a ocurrir siempre, pues para valores grandes de k tenemos que $T_k(n) < T_{k+1}(n)$.

El punto donde se alcanza el mínimo de tal sucesión de funciones va a depender de la implementación que se realice del procedimiento general, pero si se sigue un esquema similar al que nosotros hemos implementado aquí, la constante s resulta ser del orden de ocho, alcanzándose el mínimo para $k = 3$.

Entonces, ¿por qué no se enseña este método a los alumnos en vez del Mezcla clásico? La respuesta viene una vez más dada por la evaluación de la ganancia que se obtiene (en cuanto a tiempo de ejecución) frente a las desventajas de este nuevo método respecto a la dificultad y mantenimiento del código obtenido. La naturalidad y claridad del primero lo hacen preferible.

Solución al Problema 2.14. (☺)

Calculando el número de operaciones elementales que realiza el algoritmo obtenemos la ecuación en recurrencia:

$$T(n) = 22 + 3T(2n/3).$$

Esta ecuación es fácil de resolver si hacemos el cambio $n = (3/2)^k$, mediante el cual obtenemos

$$t_k = 22 + 3t_{k-1},$$

ecuación en recurrencia no homogénea con ecuación característica $(x-3)(x-1) = 0$, y por tanto la solución es:

$$t_k = c_1 3^k + c_2.$$

Deshaciendo el cambio, obtenemos finalmente:

$$T(n) = c_1 3^{\log_{3/2} n} + c_2 = c_1 n^{\log_{3/2} 3} + c_2.$$

Para el cálculo de las constantes, tomaremos dos condiciones iniciales: $T(1) = 6$ y $T(2) = 13$. Con ambas es fácil ver que $c_2 = 6$ y $c_1 = 1.07017$. Como $c_1 > 0$ podemos afirmar que:

$$T(n) \in \Theta(n^{\log_{3/2} 3}).$$

Ahora bien, $\log_{3/2}3 = 2.7095113$, con lo cual este método resulta ser de un orden de complejidad muy superior al del resto de los métodos de ordenación vistos en el presente capítulo y por tanto no rentable frente a ellos.

Solución al Problema 2.15. (☺)

Este problema es una generalización del que intenta encontrar la mediana de un vector dado (para $k = (n+1)\div 2$), conocido también como el problema de la *Selección*. Sin embargo hemos preferido referirnos a él como el problema del *k-ésimo elemento* para no confundirlo con el algoritmo de ordenación del mismo nombre.

- Efectivamente, una primera idea consiste en ordenar el vector $a[1..n]$ por algún método eficiente y luego escoger el elemento $a[k]$. Sabemos ya que este procedimiento es de complejidad $O(n\log n)$.
- Si decidiéramos modificar el procedimiento de Selección de forma que parase cuando hubiera ordenado hasta la posición k conseguiríamos cierta mejora para algunos casos, pues este algoritmo sería de complejidad $O(nk)$.
- Otra idea interesante es la de utilizar el método de ordenación por montículos, modificándolo como en el caso anterior para que pare cuando tenga ordenado hasta la posición k. Así logramos hacerlo mejor para algunos valores de k, pues este procedimiento es de complejidad $(n-k)\log n$ (ya que el método de montículos ordena de atrás hacia adelante). También podemos usar una variante en donde los montículos están ordenados de menor a mayor. Con esto conseguimos un procedimiento de complejidad $O(k\log n)$.
- ¿Puede usarse una modificación de Quicksort para resolver este problema? Observando cómo funciona este método, nos damos cuenta que la función

Pivote nos puede ayudar: tras su ejecución ha modificado el vector de forma que los elementos anteriores a la posición *l* que nos devuelve son todos menores o iguales que a[*l*], y los posteriores a *l* son mayores que a[*l*]. La idea es pues invocar repetidamente a la función *Pivote* hasta que la posición *l* coincida con la que buscamos (*k*). Con este procedimiento no es necesario realizar ninguna ordenación explícita:

```
PROCEDURE Kesimo(VAR a:vector;prim,ult,k:CARDINAL):INTEGER;
   VAR l:CARDINAL;
BEGIN
   IF prim<ult THEN
      l:=Pivote(a,a[prim],prim,ult);
      IF l>(prim+k-1) THEN RETURN Kesimo(a,prim,l-1,k) END;
      IF l<(prim+k-1) THEN RETURN Kesimo(a,l+1,ult,k-l+prim-1)
END;
      RETURN a[l]
   ELSE
      RETURN a[ult]
   END
END Kesimo;
```

Con este método conseguimos resolver el problema, y tiene un orden de complejidad lineal para la mayoría de los casos, lo cual es mejor de lo conseguido hasta ahora.

Sin embargo, por estar basado en Quicksort, este método va a heredar sus casos conflictivos: cuando el vector está ya ordenado o cuando todos los elementos del vector son iguales. En ambos casos nuestro procedimiento resulta ineficiente, pues su complejidad es cuadrática.

¿Podemos corregir de alguna forma estos casos extremos? Nuestra siguiente idea es obvia tras haber realizado el problema 2.8. Podríamos usar la función *Pivote2* para eliminar el caso en que todos los elementos son iguales, con lo que obtendríamos:

```
PROCEDURE Kesimo2(VAR a:vector;prim,ult,k:CARDINAL):INTEGER;
   VAR i,d:CARDINAL;
BEGIN
   IF prim<ult THEN
      Pivote2(a,a[prim],prim,ult,i,d);
      IF (prim+k-1)<i THEN RETURN Kesimo2(a,prim,i-1,k) END;
      IF d<=(prim+k-1) THEN RETURN Kesimo2(a,d,ult,k-d+prim) END;
      RETURN a[i]
   ELSE
      RETURN a[ult]
   END
END Kesimo2;
```

Sin embargo, esta función deja pendiente el caso en que el vector esté ordenado (de forma creciente o decreciente), pues escogemos como pivote el

primer elemento del vector. Lo mejor sería escoger la mediana, que es el elemento que queda justo en medio, asegurándonos así que en cada iteración dividimos por dos los elementos a considerar; esto conllevaría un tiempo de ejecución de orden lineal.

El problema es que esto parece contradictorio, puesto que hemos reducido el problema de calcular la mediana a ¡calcular la mediana! ¿Cómo salimos ahora de esta situación?

Una posible solución puede encontrarse en [BRA97], y se basa en una técnica bastante general, y por eso hemos querido incluir aquí este tipo de problemas. Basta con encontrar una función que calcule una mediana "aproximada" y a partir de ahí utilizar el algoritmo *Kesimo* con $k = (n+1) \div 2$.

```
PROCEDURE Kesimo3(VAR a:vector;prim,ult,k:CARDINAL):INTEGER;
   VAR i,d:CARDINAL; pm:INTEGER; (* pseudo_mediana *)
BEGIN
   IF prim<ult THEN
      pm:=CasiMediana(a,prim,ult);
      Pivote2(a,pm,prim,ult,i,d);
      IF (prim+k-1)<i THEN RETURN Kesimo3(a,prim,i-1,k) END;
      IF d<=(prim+k-1) THEN RETURN Kesimo3(a,d,ult,k-d+prim) END;
      RETURN a[i]
   ELSE
      RETURN a[ult]
   END
END Kesimo3;
```

La función *CasiMediana* es la que calcula la mediana aproximada, y se basa en otra función auxiliar, *Medianade5*, que determina la mediana de un vector de 5 elementos:

```
PROCEDURE CasiMediana(VAR a:vector; prim,ult:CARDINAL):INTEGER;
(* calcula una mediana aproximada del vector a[prim..ult] *)
   VAR n,i:CARDINAL; b:vector;
BEGIN
   n:=ult-prim+1;
   IF n<=5 THEN
      RETURN Medianade5(a,prim,ult)
   END;
   n:=n DIV 5;
   FOR i:=1 TO n DO
      b[i]:=Medianade5(a,5*i-4+prim-1,5*i+prim-1)
   END;
   RETURN Kesimo3(b,1,n,(n+1)DIV 2)
END CasiMediana;

PROCEDURE Medianade5(VAR a:vector; prim,ult:CARDINAL):INTEGER;
```

```
(* calcula la mediana de un vector de hasta 5 elementos (es
   decir, ult<prim+5) *)
   VAR i,n:CARDINAL; b:vector; (* para no modificar el vector *)
BEGIN
   n:=ult-prim+1; (* numero de elementos *)
   FOR i:=1 TO n DO
      b[i]:=a[prim+i-1]
   END;
   FOR i:=1 TO (n+1) DIV 2 DO
      Intercambia(b,i,PosMinimo(b,i,n))
   END;
   RETURN b[(n+1) DIV 2];
END Medianade5;
```

El procedimiento es capaz de calcular el k-ésimo elemento de un vector en tiempo lineal respecto al tamaño de la entrada, aunque sin embargo vuelve a ocurrir aquí lo que comentamos anteriormente. Para conseguir un tiempo lineal hemos tenido que pagar un precio, que en este caso es que la constante multiplicativa del tiempo de ejecución de este método se ha visto duplicada.

La decisión de si merece la pena pagar ese precio sólo para cubrir los dos casos especiales del algoritmo la dejamos al usuario del mismo.

Solución al Problema 2.16. (☺)

a) La opción de ordenar el vector y escoger los m primeros elementos es de complejidad $O(n\log n)$ si escogemos uno de los métodos de ordenación de este orden.
b) Si usamos repetidamente el procedimiento de ordenación por Selección conseguimos una mejora en la complejidad para valores pequeños de m: el método es de orden $O(mn)$.
c) Como el procedimiento que encuentra el k-ésimo elemento es de complejidad lineal, invocándolo m veces obtenemos también un método de orden $O(mn)$.

Estudiemos por tanto otras opciones.

- Pensando en modificar alguno de los métodos de ordenación ya conocidos, podemos pensar en el método de ordenación por montículos. Es fácil modificar este método para conseguir un procedimiento que encuentre los m elementos más pequeños en tiempo $O((n-m)\log n)$, o bien en tiempo $O(m\log n)$ si utilizamos montículos invertidos, es decir, en donde en la raíz se encuentra el menor elemento.
- Pero es al pensar en una modificación de Quicksort cuando conseguimos un algoritmo basado en su estrategia y con mejor tiempo. Queremos buscar los m elementos menores de un vector, y podemos utilizar la función *Pivote* para esto.

Así, nuestro propósito es ir dejando a la izquierda del pivote los elementos menores que él hasta que consigamos que nuestro pivote sea el m-ésimo elemento:

```
PROCEDURE Menores(VAR a:vector;prim,ult,m:CARDINAL);
   VAR l:CARDINAL;
BEGIN
   IF prim<ult THEN
      l:=Pivote(a,a[prim],prim,ult);
      IF l>(prim+m-1) THEN Menores(a,prim,l-1,m)
      ELSIF l<(prim+m-1) THEN Menores(a,l+1,ult,m-l+prim-1)
      END
   END
END Menores;
```

Obsérvese que en las m primeras posiciones del vector se encuentran todos los elementos pedidos, aunque no necesariamente ordenados, por la forma como trabaja la función *Pivote*. Este procedimiento es, como en el caso de buscar el k-ésimo elemento, de complejidad lineal.

Como mencionábamos en el problema anterior, esta solución es de orden lineal menos en dos casos, ambos heredados del método original de Quicksort. Cuando el vector está ordenado de antemano y cuando todos los elementos del vector son iguales. Ante estas dos circunstancias nuestro procedimiento ofrece un comportamiento de complejidad cuadrática. Al igual que entonces, podemos aplicar los mecanismos vistos en el problema anterior que permiten al método tener en cuenta estos dos casos, aunque pagando cierto precio en los demás.

Solución al Problema 2.17. (😐)

Volvemos a encontrarnos aquí con un caso similar a los tratados en los dos problemas anteriores. Las primeras opciones ya son conocidas:

a) La opción de ordenar el vector y escoger los m elementos pedidos es de complejidad $n\log n$ si escogemos uno de los métodos de ordenación de este orden.

b) Si usamos repetidamente el procedimiento de ordenación por Selección conseguimos una complejidad de orden $O((m+n/2)n)$, siendo este caso peor que la primera opción (por ser cuadrática).

c) Como el procedimiento que encuentra el k-ésimo elemento es de complejidad lineal, invocándolo m veces obtenemos un método de orden $O(mn)$.

d) Usando una modificación del método de Montículos obtendríamos un procedimiento de orden $O((m+n/2)\log n)$.

¿Cómo podríamos utilizar la estrategia de Quicksort para conseguir un método mejor?

- La primera idea consiste en acotar, usando un procedimiento análogo al del problema anterior, los elementos $n÷2$ y $(n÷2)+m–1$. Esto nos dejaría en medio de ambos los elementos buscados.
- Sin embargo, existe un método mejor, cuya idea consiste en localizar el elemento $n÷2$ mediante el procedimiento del *k*-ésimo (problema 2.15), y luego utilizar el procedimiento *Menores* del problema anterior (2.16) sobre el subvector [*n*÷2..*ult*]:

```
PROCEDURE Medios(VAR a:vector;prim,ult,l,m);
  VAR x:INTEGER;
BEGIN
  x:=Kesimo(a,prim,ult,l);
  Menores(a,l,ult,m)
END Medios;
```

El procedimiento que hemos implementado aquí es un poco más general, puesto que busca los elementos que ocuparían las posiciones l, $l+1,...,m$, con $l < m$. Basta invocarlo con $l = (n÷2)$ para obtener el método pedido. Y como puede observarse, la complejidad de este método es lineal, por serlo cada uno de los procedimientos que lo componen.